Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence ••• (••••) •••–•••
Artificial Intelligence
www.elsevier.com/locate/artint

State-set branching: Leveraging BDDs for heuristic search ☆

Rune M. Jensen *, Manuela M. Veloso, Randal E. Bryant

Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213-3891, USA

Received 22 February 2006; received in revised form 30 May 2007; accepted 30 May 2007

Abstract

In this article, we present a framework called state-set branching that combines symbolic search based on reduced ordered Binary Decision Diagrams (BDDs) with best-first search, such as A* and greedy best-first search. The framework relies on an extension of these algorithms from expanding a single state in each iteration to expanding a set of states. We prove that it is generally sound and optimal for two A* implementations and show how a new BDD technique called branching partitioning can be used to efficiently expand sets of states. The framework is general. It applies to any heuristic function, evaluation function, and transition cost function defined over a finite domain. Moreover, branching partitioning applies to both disjunctive and conjunctive transition relation partitioning. An extensive experimental evaluation of the two A* implementations proves state-set branching to be a powerful framework. The algorithms outperform the ordinary A* algorithm in almost all domains. In addition, they can improve the complexity of A* exponentially and often dominate both A* and blind BDD-based search by several orders of magnitude. Moreover, they have substantially better performance than BDDA*, the currently most efficient BDD-based implementation of A*.

Keywords: Heuristic search; BDD-based search; Boolean representation

1. Introduction

Informed or *heuristic* best-first search (BFS) algorithms[1] such as greedy best-first search and A* [27] are considered important contributions of AI. The advantage of these algorithms, compared to *uninformed* or *blind* search algorithms such as depth-first search and breadth-first search, is that they use heuristics to guide the search toward the goal and in this way significantly reduce the number of visited states. The algorithms differ mainly by the way they evaluate nodes in the search tree. A* is probably the most widely known BFS algorithm. Each search node of A* is associated

☆ This work is an extended version of a paper presented at AAAI-02 [R.M. Jensen, R.E. Bryant, M.M. Veloso, SetA*: An efficient BDD-based heuristic search algorithm, in: Proceedings of 18th National Conference on Artificial Intelligence (AAAI-02), 2002, pp. 668–673]. The work was supported in part by the Danish Research Agency and the United States Air Force under Grants Nos F30602-00-2-0549 and F30602-98-2-0135. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force, or the US Government.

* Corresponding author.
E-mail addresses: runej@cs.cmu.edu (R.M. Jensen), mmv@cs.cmu.edu (M.M. Veloso), bryant@cs.cmu.edu (R.E. Bryant).
URLs: http://www.cs.cmu.edu/~runej (R.M. Jensen), http://www.cs.cmu.edu/~mmv(M.M. Veloso), http://www.cs.cmu.edu/~bryant (R.E. Bryant).

[1] In this article, BFS always refers to best-first search and not breadth-first search.

0004-3702/$ – see front matter
doi:10.1016/j.artint.2007.05.009

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

with a cost g of reaching the node and a heuristic estimate h of the remaining cost of reaching the goal. In each iteration, A* expands a node with minimum expected completion cost $f = g + h$. A* can be shown to have much better performance than uninformed search algorithms. However, an unresolved problem for this algorithm is that the number of expanded search nodes may grow exponentially even if the heuristic has only a small constant relative error [46]. Such heuristic functions are often encountered in practice, since many heuristics are derived from a relaxation of the search problem that is likely to introduce a relative error. Furthermore, in order to detect duplicate states and construct a solution, A* must keep all expanded nodes in memory. For this reason, the limiting factor of A* is often space rather than time.

In *symbolic model checking* [42], a quite different approach has been taken to verify systems with large state spaces. Instead of representing and manipulating sets of states explicitly, this is done implicitly using Boolean functions.[2] Given a bit vector encoding of states, *characteristic functions* are used to represent subsets of states. In a similar way, a Boolean function can be used to represent the transition relation of a domain and find successor states via Boolean function manipulation. The approach potentially reduces both the time and space complexity exponentially. Indeed during the last decade, remarkable results have been obtained using reduced ordered *Binary Decision Diagrams* (BDDs [9]) as the Boolean function representation. Systems with more than 10^{100} states have been successfully verified with the BDD-based model checker SMV [42]. For several reasons, however, only very limited work on using heuristics to guide these implicit search algorithms has been carried out. First of all, the solution techniques considered in formal verification often require traversal of all reachable states making search guidance irrelevant. Secondly, it is non-trivial to efficiently handle cost estimates such as the g and h-costs associated with individual states when representing states implicitly.

In this article, we present a new framework called *state-set branching* that combines BDD-based search and best-first search (BFS) and efficiently solves the problem of representing cost estimates. State-set branching applies to any BFS algorithm and any transition cost function, heuristic function, and node-evaluation function defined over a finite domain. The state-set branching framework consists of two independent parts. The first part extends a general BFS algorithm to an algorithm called best-set-first search (BSFS) that expands sets of states in each iteration. The second part is an efficient BDD-based implementation of BSFS using a partitioning of the transition relation of the search domain called *branching partitioning*. Branching partitioning allows sets of states to be expanded implicitly and sorted according to their associated cost estimates. The approach applies both to disjunctive and conjunctive partitioning [15].

Two implementations of A* based on the state-set branching framework called FSETA* anf GHSETA* have been experimentally evaluated in 10 search domains ranging from VLSI-design with synchronous actions, to classical AI planning problems such as the $(N^2 - 1)$-puzzles and problems used in the international planning competitions 1998–2004 [2,29,39,40]. We apply four different families of heuristic functions ranging from the minimum Hamming distance to the sum of Manhattan distances for the $(N^2 - 1)$-puzzles, and HSPr [8] for the planning problems. In this experimental evaluation, the two A* implementations outperform implementations of the ordinary A* algorithm in all domains except one where an efficient Boolean state encoding seems to be challenging to find.[3] In addition, the results show that they can improve the complexity of A* exponentially and that they often dominate both the ordinary A* algorithm and blind BDD-based search by several orders of magnitude. Moreover, they have substantially better performance than BDDA*, the currently most efficient symbolic implementation of A*.

The main limitation of the state-set branching framework is that a Boolean state encoding with a compact BDD representation must be found for a target domain. In most cases this is easy, but for general domain representation languages such as PDDL [24] it may be challenging to define automated encoding techniques. Another issue is whether branching partitionings are easy to obtain for all heuristics. The experiments in this article show that additive heuristics like the sum of Manhattan distances and the HSPr heuristic can be represented compactly. A recent study [32], however, shows that branching partitionings of the max-pair heuristic [28] may be prohibitively large. It is not our impression, though, that strong domain dependent heuristics are as combinatorial complex as the max-pair heuristic.

[2] By an explicit representation, we mean an enumerative representation that uses space linear in the number of represented elements. By an implicit representation, we mean a non-enumerative representation using Boolean expressions to characterize elements.

[3] By ordinary A* we refer to the graph-search version of A* that maintains a closed list for duplicate elimination and uses an explicit state representation.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

The remainder of this article is organized as follows. We first describe related work in Section 2. We then define search problems in Section 3 and describe the general BFS algorithm in Section 4. In Section 5, we extend this algorithm to expand sets of states and study a number of example applications of the new best-set-first search algorithm. In Section 6, we introduce branching partitioning and other BDD-based techniques to efficiently implement these algorithms. The experimental evaluation is described in Section 7. Finally, we conclude and discuss directions for future work in Section 8.

2. Related work

State-set branching is the first general framework for combining heuristic search and BDD-based search. All previous work has been restricted to particular algorithms. BDD-based heuristic search has been investigated independently in symbolic model checking and AI. The pioneering work is in symbolic model checking where heuristic search has been used to falsify design invariants by finding error traces. Yuan et al. [60] study a bidirectional greedy best-first search algorithm pruning frontier states according to their minimum Hamming distance[4] to error states. BDDs representing Hamming distance equivalence classes are precomputed and conjoined with BDDs representing the search frontier during search. Yang and Dill [59] also consider minimum Hamming distance as heuristic function in an ordinary greedy best-first search algorithm. They develop a specialized BDD operation for sorting a set of states according to their minimum Hamming distance to a set of error states. The operation is efficient with linear complexity in the size of the BDD representing the error states. However, it is unclear how such an operation can be generalized to other heuristic functions. In addition, this approach finds next states and sorts them according to their cost estimates in two separate phases. Recent applications of BDD-based heuristic search in symbolic model checking include error directed search [51] and using symbolic pattern databases for guided invariant model checking [49].

In general, heuristic BDD-based search has received little attention in symbolic model checking. The reason is that the main application of BDDs in this field is verification where all reachable states must be explored. For Computation Tree Logic (CTL) checking [15], guiding techniques have been proposed to avoid a blow-up of intermediate BDDs during a reachability analysis [7]. However, these techniques are not applicable to search since they are based on defining lower and upper bounds of the fixed-point of reachable states.

In AI, Edelkamp and Reffel [21] developed the first BDD-based implementation of A* called BDDA*. BDDA* can use any heuristic function defined over a finite domain and has been applied to planning as well as model checking [51]. Several extensions of BDDA* have been published including duplicate elimination, weighted evaluation function, pattern data bases, disjunctive transition relation partitioning, and external storage [18,19]. BDDA* is currently the most efficient symbolic implementation of A*. It contributes a combination of A* and BDDs where a single BDD is used to represent the search queue of A*. In each iteration, all states with minimum f-costs are extracted from this BDD. The successor states and their associated f-cost are then computed via arithmetic BDD operations and added to the BDD representing the search queue.

There are two major differences between BDDA* and the SETA* algorithms presented in this article.

(1) Our experimental evaluation of BDDA* shows that its successor state function scales poorly (see Section 7.6). A detailed analysis of the computation shows that the complexity mainly is due to the symbolic arithmetic operations. For this reason, a main philosophy of state-set branching is to use BDDs only to represent state information. Cost estimates like the f-cost of a state is represented explicitly in a search tree.
(2) State-set branching introduces a novel approach called *branching partitioning* that makes it possible to use a transition relation partitioning to propagate cost estimates efficiently between *sets* of states during search. In this way, a best-first search algorithm called *best-set-first search* that expands sets of states in each iteration can be efficiently implemented with BDDs. As shown by our experimental evaluation in Section 7, this has a dramatic positive effect on the efficiency of the algorithms.

An ADD-based[5] implementation of A* called ADDA* has also been developed [26]. ADDs [3] generalize BDDs to finite valued functions and may simplify the representation of numeric information like the f-cost of states [58].

[4] The Hamming distance between two Boolean vectors is the number of bits in the two vectors with different value.
[5] ADD stands for Algebraic Decision Diagram [3].

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Similar to BDDA*, however, ADDA* performs arithmetic computations via complex ADD operations. It has not been shown to have better performance than BDDA* [26].

A recent comparison of A* and a symbolic implementation of A* called SA* on 500 random 8-puzzle problems shows that SA* consistently uses more memory than A* and is outperformed by A* if the heuristic is strong [45]. These results are not confirmed by the experimental evaluation in this article where GHSETA* typically uses less memory than A* and often finds solutions much faster than A*. We believe that there are several reasons for the observed differences. First, SA* does not use state-set branching to compute child nodes but instead relies on the less efficient two phase approach developed by Yuan et al. Second, SA* stores expanded nodes without merging nodes with the same g-cost. This is done by GHSETA* and may lead to significant space savings. Third, 8-puzzle problems are very small ($<10^6$ states) compared with the benchmark problems considered in our evaluation. It is unclear to what extent symbolic approaches pay off on such small problems. Fourth, the state-space of an (N^2-1)-puzzle is a subspace of a permutation space consisting of all possible permutations of n elements. It is easy to show that a BDD representation of a permutation space is exponentially more compact than an explicit representation. It is, however, still exponential in the number of elements in the permutation. For this reason, we may expect a high memory consumption of BDD-based search on (N^2-1)-puzzles. Indeed, we get fairly weak results for FSETA* and GHSETA* on the 24- and 35-puzzle benchmarks.

Other related applications of BDDs for search include HTN planning [37], STRIPS planning [10,12,16,23,33, 56], universal planning [13,34], adversarial planning [17,35], fault tolerant planning [36], conformant planning [14], planning with extended goals [47], planning under partial observability [5,6], and shortest path search [52,53].

3. Search problems

A search domain is a finite graph where vertices denote world states and edges denote state transitions. Transitions are caused by activity in the world that changes the world state deterministically. Sets of transitions may be defined by *actions*, *operator schemas*, or *guarded commands*. In this article, however, we will not consider such abstract transition descriptions. If a transition is directed from state s to state s', state s' is said to be a *successor* of s and state s is said to be the *predecessor* of s'. The number of successors emanating from a given state is called the *branching factor* of that state. Since the domain is finite, the branching factor of each state is also finite. Each transition is assumed to have positive *transition cost*.

Definition 1 *(Search domain).* A search domain is a triple $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation, and $c : \mathcal{T} \to \mathbb{R}^+$ is a transition cost function.

A search problem is a search domain with a single initial state and a set of goal states.

Definition 2 *(Search problem).* Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ be a search domain. A search problem for $\mathcal{D}$ is a triple $\mathcal{P} = \langle \mathcal{D}, s_0, \mathcal{G} \rangle$ where $s_0 \in \mathcal{S}$ and $\mathcal{G} \subseteq \mathcal{S}$.

A solution π to a search problem is a path from the initial state to one of the goal states. The *solution length* is the number of transitions in π and the *solution cost* is the sum of the transition costs of the path.

Definition 3 *(Search problem solution).* Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ be a search domain and $\mathcal{P} = \langle \mathcal{D}, s_0, \mathcal{G} \rangle$ be a search problem for $\mathcal{D}$. A solution to $\mathcal{P}$ is a sequence of states $\pi = s_0, \ldots, s_n$ such that $s_n \in \mathcal{G}$, and $\mathcal{T}(s_j, s_{j+1})$ for $j = 0, 1, \ldots, n-1$.

An *optimal solution* to a search problem is a solution with minimum cost. We will use the symbol C^* to denote the minimum cost. Fig. 1 shows a search problem example and an optimal solution.

4. Best-first search

Best-first search algorithms are characterized by building a search tree superimposed over the state space during the search process. Each *search node* in the tree is a pair $\langle s, \vec{e} \rangle$ where s is a single state and $\vec{e} \in \mathbb{R}^d$ is a d-dimensional real vector representing the cost estimates associated with the node (e.g., $\vec{e}$ could be a two dimensional vector containing

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

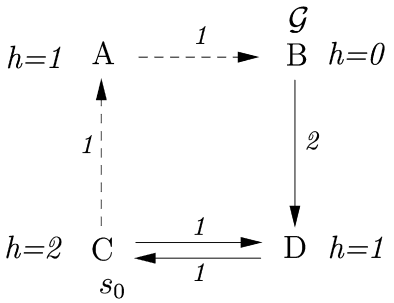

Fig. 1. An example search problem consisting of four states, five transitions, initial state $s_0 = \mathrm{C}$, and a single goal state $\mathcal{G} = \{B\}$. The dashed path is an optimal solution. The h-costs associated with each state define the heuristic function used in Section 4.

```
function BFS(s_0, ē_0, G)
1   frontier ← MAKEQUEUE(⟨s_0, ē_0⟩)
2   loop
3     if |frontier| = 0 then return failure
4     ⟨s, ē⟩ ←REMOVETOP(frontier)
5     if s ∈ G then return EXTRACTSOLUTION(frontier, ⟨s, ē⟩)
6     frontier ← ENQUEUEALL(frontier, EXPAND(⟨s, ē⟩))
```

Fig. 2. The general best-first search algorithm.

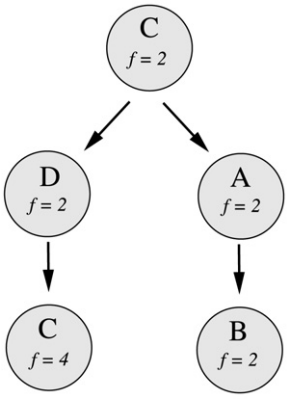

Fig. 3. Search tree example.

the g and h-cost associated with a search node of A*). Fig. 2 shows a general BFS algorithm. We assume that the initial state is associated with cost estimate $\vec{e}_0$. The solution extraction function in line 5 simply obtains a solution by tracing back the transitions from the goal node to the root node. EXPAND in line 6 finds the set of child nodes of a single node, and ENQUEUEALL inserts each child in the frontier queue.

A* is a BFS algorithm[6] that sorts the unexpanded nodes in the priority queue in ascending order of the a cost estimate given by a *heuristic evaluation function* f. The evaluation function is defined by

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the cost of the path in the search tree leading from the root node to n, and $h(n)$ is a *heuristic function* estimating the cost of a minimum cost path leading from the state in n to some goal state.[7] Thus $f(n)$ measures the minimum cost over all solution paths constrained to go through the state in n. The search tree built by A* for the example problem and heuristic function defined in Fig. 1 is shown in Fig. 3.

[6] However, practical implementations of A* includes a *closed list* to detect duplicate states.

[7] For a heuristic function to be valid, we require that $h(n) \geqslant 0$ for all n and $h(n) = 0$ for all n containing a goal state.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

A* is *sound* and *complete*, since the node expansion operation is assumed to be correct, and infinite cyclic paths have unbounded cost. A* further finds optimal solutions if the heuristic function $h(n)$ is *admissible*, that is, if $h(n) \leqslant h^*(n)$ for all n, where $h^*(n)$ is the minimum cost of a path going from the state in n to a goal state. The heuristic function is called *consistent* if $h(n) \leqslant c(n, n') + h(n')$ for every successor node n' of n. The complexity of A* is directly tied to the accuracy of the estimates provided by h. When A* employs a perfectly informed heuristic ($h(n) = h^*(n)$) and f-cost ties are broken by giving highest priority to the node with lowest h-cost, it is guided directly toward the closest goal. At the other extreme, when no heuristic at all is available ($h(n) = 0$), the search becomes exhaustive, normally yielding exponential complexity. In general, A* with duplicate elimination using a consistent heuristic has linear complexity if the absolute error of the heuristic function is constant, but it may have exponential complexity if the relative error is constant. Subexponential complexity requires that the growth rate of the error is logarithmically bounded [46]

$$\big|h(n) - h^*(n)\big| \in \mathrm{O}\big(\log h^*(n)\big).$$

The complexity results are discouraging due to the fact that practical heuristic functions often are based on a relaxation of the search problem that causes $h(n)$ to have constant or near constant relative error. The results show that practical application of A* still may be very search intensive. Often better performance of A* can be obtained by weighting the g- and h-component of the evaluation function [48]

$$f(n) = (1 - w)g(n) + wh(n), \quad \text{where } w \in [0, 1]. \tag{1}$$

Weights $w = 0.0, 0.5$, and 1.0 correspond to uniform cost search, A*, and greedy best-first search. Weighted A* is optimal in the range $[0.0, 0.5]$ if the heuristic function is admissible but often finds solutions faster in the range $(0.5, 1]$.

5. State-set branching

The state-set branching framework has two independent parts: a modification of the general BFS algorithm to a new algorithm called *best-set-first search* (BSFS), and a collection of BDD-based techniques for implementing the new algorithm efficiently. In this section, we will describe the BSFS algorithm. In the next section, we show how it is implemented with BDDs.

5.1. Best-set-first search

Assume that each transition $\mathcal{T}(s, s')$ for a particular heuristic search algorithm changes the cost estimates with $\delta\vec{e}(s, s')$. Thus if s is associated with cost estimates $\vec{e}$ and s' is reached by $\mathcal{T}(s, s')$ then s' will be associated with cost estimates $\vec{e} + \delta\vec{e}(s, s')$. For A*, the cost estimates can be one or two dimensional: either it is the f-cost or the g and h-cost of a search node. In the first case $\delta\vec{e}(s, s')$ is the f-cost change caused by the transition. The δf costs of our example problem are shown in Fig. 4. The BSFS algorithm shown in Fig. 5 is almost identical to the BSFS algorithm defined in Fig. 2. However, the state set version traverses a search tree during the search process where each search node contains a set of states associated with the same cost estimates. Multiple states in each node emerge because child nodes having identical cost estimates are coalesced by STATESETEXPAND in line 6 and because

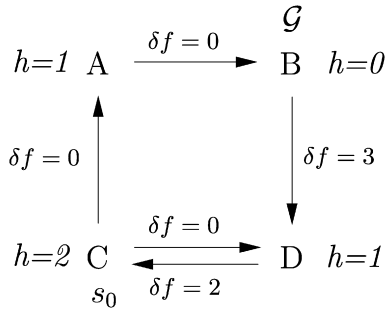

Fig. 4. The example search problem with δf costs.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

```
function BSFS(s_0, e⃗_0, 𝒢)
1   frontier ← MAKEQUEUE(({s_0}, e⃗_0))
2   loop
3     if |frontier| = 0 then return failure
4     ⟨S, e⃗⟩ ← REMOVETOP(frontier)
5     if S ∩ 𝒢 ≠ ∅ then return EXTRACTSOLUTION(frontier, ⟨S ∩ 𝒢, e⃗⟩)
6     frontier ← ENQUEUEANDMERGE(frontier, STATESETEXPAND(⟨S, e⃗⟩))
```

Fig. 5. The best-set-first search algorithm.

```
function STATESETEXPAND(⟨S, e⃗⟩)
1   child ← emptyMap
2   foreach state s in S
3     foreach transition 𝒯(s, s′)
4       e⃗_c ← e⃗ + δe⃗(s, s′)
5       child[e⃗_c] ← child[e⃗_c] ∪ {s′}
6   return MAKENODES(child)
```

Fig. 6. The state set expand function.

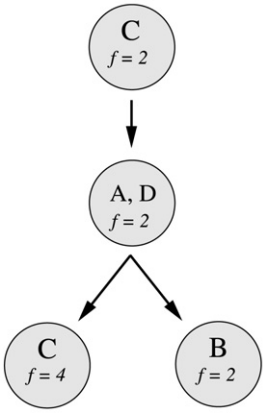

Fig. 7. State-set search tree example.

ENQUEUEANDMERGE may merge child nodes with nodes on the *frontier* queue having identical cost estimates. The state-set expansion function is defined in Fig. 6. The next states of some child associated with cost estimates $\vec{e}$ are stored in $child[\vec{e}]$. The outgoing transitions from each state in the parent node are used to find all successor states. The function MAKENODES called at line 6 constructs the child nodes from the completed child map. Each child node contains states having identical cost estimates. However, there may exist several nodes with the same cost estimates. In addition, MAKENODES may prune some of the child states (e.g., to implement duplicate elimination in A*).

As an example, Fig. 7 shows the search tree traversed by the BSFS algorithm for A* applied to our example problem. In order to reduce the number of search nodes even further, ENQUEUEANDMERGE of the BSFS algorithm may merge nodes on the search frontier having identical cost estimates. This transforms the search tree into a *Directed Acyclic Graph* (DAG), but as proven in Appendix A this does not affect the soundness of the BSFS algorithm. The EXTRACTSOLUTION function in line 5 uses the backward traversal described in the proof of Lemma 7 to extract a solution. It is not possible to show completeness of the BSFS algorithm since it covers incomplete algorithms such as greedy best-first search.

5.2. *The* FSETA* *and* GHSETA* *algorithms*

The BSFS algorithm can be used to implement variants of greedy best-first search, A*, weighted A*, uniform cost search, and beam search. To simplify the presentation of BSFS, we have only described its tree-search version, where states may be repeated many times in the search tree. In a concrete application of BSFS, however, a closed

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

list of expanded states is maintained to eliminate many of these duplicates. The elimination strategy depends on the application and will be discussed independently for each algorithm below.[8]

Greedy best-first search is implemented by using the values of the heuristic function as cost estimates and sorting the nodes on the frontier in ascending order, such that the top node contains states with least h-cost. The cost estimate of the initial state is $\vec{e}_0 = h(s_0)$ and each transition $\mathcal{T}(s, s')$ is associated with the change in h, that is, $\delta\vec{e}(s, s') = h(s') - h(s)$. In each iteration, this greedy best-first search algorithm will expand all states with least h-cost on the frontier. A strategy for eliminating duplicates that does not compromise completeness is to subtract all the states in the closed list from the set of states to expand.

A* can be implemented by setting $\vec{e}_0 = h(s_0)$ and $\delta\vec{e}(s, s') = c(s, s') + h(s') - h(s)$ such that the cost estimates equal the f-cost of search nodes. Again nodes on the frontier are sorted in ascending order and the node with least f-cost is expanded in each iteration. If the heuristic is consistent, a strategy for eliminating duplicates that does not compromise optimality is to subtract all the states in the closed list from the set of states to expand. However, since this is not possible in general for admissible heuristics, we consider an implementation without duplicate elimination called FSETA*. The FSETA* algorithm always merges nodes on the frontier associated with the same f-cost.[9]

An A* implementation with duplicate elimination that does not require the heuristic function to be admissible or consistent must keep track of the g and h-cost separately and prune child states reached previously with a lower g-cost. To achieve this, we can define $\vec{e}_0 = (0, h(s_0))$ and $\delta\vec{e}(s, s') = (c(s, s'), h(s') - h(s))$. The frontier is, as usual, sorted according to the evaluation function $f(n) = g(n) + h(n)$. An implementation that uses the above strategy for eliminating duplicates is called GHSETA*. Compared to FSETA*, GHSETA* merges nodes that have identical g and h-costs. Thus there may be several nodes on the frontier with same f-cost but different g and h-costs. In each iteration, GHSETA* may therefore only expand a subset of the states on the frontier with minimum f-cost. A number of other improvements have been integrated in GHSETA*. First, it applies the usual tie breaking rule for nodes with identical f-cost choosing the node with the least h-cost. Thus, in situations where all nodes on the frontier have $f(n) = C^*$, the algorithm focuses the search in a DFS fashion. The reason is that a node at depth level d in this situation must have greater h-cost than a node at level $d + 1$ due to the non-negative transition costs. In addition, it only merges two nodes on the frontier if the space used by the resulting node is less than an upper-bound u. This may help to focus the search further in situations where there is an abundance of solutions, but space requirements of the frontier nodes grow fast with the search depth.

Both GHSETA* and FSETA* can easily be extended to the weighted A* algorithm described in Section 4. Using an approach similar to the one used by Pearl [46], FSETA* and GHSETA* can be shown to be optimal given an admissible heuristic. In particular this is true when using the trivial admissible heuristic function $h(n) = 0$ of uniform cost search. The proofs are given in Appendix A.[10]

6. BDD-based implementation

The motivation for defining the BSFS algorithm is that it can be efficiently implemented with BDDs. In this section, we describe how to represent sets of states implicitly with BDDs and develop a technique called *branching partitioning* for expanding search nodes efficiently.

6.1. The BDD representation

A BDD is a decision tree representation of a Boolean function on a set of linearly ordered arguments. The tree is reduced by removing redundant tests on argument variables and reusing structure. This transforms the tree into a rooted directed acyclic graph and makes the representation canonical. BDDs have several advantages: first, many functions encountered in practice (e.g., symmetric functions) have polynomial size, second, graphs of several BDDs can be shared and efficiently manipulated in multi-rooted BDDs, third, with the shared representation, equivalence and satisfiability tests on BDDs take constant time, and finally, fourth, the 16 Boolean operations on two BDDs x and y have time and space complexity $O(|x||y|)$ [9]. A disadvantage of BDDs is that there may be an exponential size

[8] The graph-search version of BSFS never re-expands a state. This strategy, however, may compromise optimality for some applications.

[9] Another reason for studying this algorithm is that it expands the same set of states as BDDA*.

[10] Notice that it follows from the optimality proof given in Appendix A that FSETA* and GHSETA* are complete.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

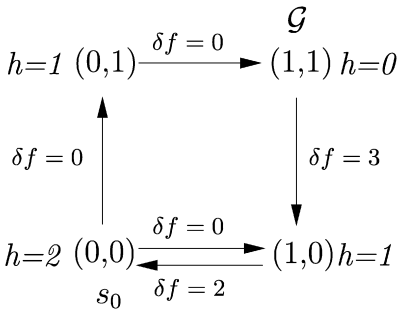

Fig. 8. Boolean state encoding of the example search problem.

difference depending on the ordering of the variables. However, powerful heuristics exist for finding good variable orderings [44]. For a detailed introduction to BDDs, we refer the reader to Bryant's original paper [9] and the books [44,58].

6.2. BDD-based state space exploration

BDDs were originally applied to digital circuit verification [1]. More relevant, however, for the work presented in this article, they were later applied in *model checking* using a range of techniques collectively coined *symbolic model checking* [42]. During the last decade BDDs have successfully been applied to verify very large transition systems. The essential computation applied in symbolic model checking is an efficient reachability analysis where BDDs are used to represent sets of states and the transition relation.

Search problems can be solved using the standard machinery developed in symbolic model checking. Let $\mathcal{D} = \langle \mathcal{S}, \mathcal{T}, c \rangle$ be a search domain. Since the number of states $\mathcal{S}$ is finite, a vector of n Boolean variables $\vec{v} \in \mathbb{B}^n$ can be used to represent the state space. In the remainder, let Z denote the set of variables in $\vec{z}$. The variables V of $\vec{v}$ are called *state variables*. A set of states S can be represented by a characteristic function $S(\vec{v})$ on $\vec{v}$. Thus, a BDD can represent any set of states. The main efficiency of the BDD representation is that the cardinality of the represented set is not directly related to the size of the BDD. For instance, the BDD of the constant function *True* has a single node and can represent all states in the domain no matter how many there are. In addition, the set operations union, intersection and complementation simply translate into disjunction, conjunction, and negation on BDDs.

In a similar way, the transition relation $\mathcal{T}$ can be represented by a characteristic function $T(\vec{v}, \vec{v}\,')$. We refer to $\vec{v}$ and $\vec{v}\,'$ as *current* and *next state variables*, respectively. To make this clear, two Boolean variables $\vec{v} = (v_0, v_1)$ are used in Fig. 8 to represent the four states of our example problem.[11] The initial state s_0 and goal state $\mathcal{G}$ are represented by two BDDs for the expressions $\neg v_0 \wedge \neg v_1$ and $v_0 \wedge v_1$, respectively. The transition relation is represented by a BDD equal to the Boolean function

$$\begin{aligned} T(\vec{v}, \vec{v}\,') = {} & \neg v_0 \wedge \neg v_1 \wedge v_0' \wedge \neg v_1' \vee \neg v_0 \wedge \neg v_1 \wedge \neg v_0' \wedge v_1' \\ & \vee \neg v_0 \wedge v_1 \wedge v_0' \wedge v_1' \vee v_0 \wedge v_1 \wedge v_0' \wedge \neg v_1' \\ & \vee v_0 \wedge \neg v_1 \wedge \neg v_0' \wedge \neg v_1'. \end{aligned}$$

The crucial idea in BDD-based or symbolic search is to stay at the BDD level when finding the next states of a set of states. A set of next states can be found by computing the *image* of a set of states S encoded in current state variables

$$\text{IMG}(S) = \big(\exists \vec{v}\,.\, S(\vec{v}) \wedge T(\vec{v}, \vec{v}')\big)[\vec{v}'/\vec{v}].$$

The previous states of a set of states is called the *preimage* and are computed in a similar fashion. The operation $[\vec{v}/\vec{v}]$ is a regular variable substitution. Existential quantification is used to abstract variables in an expression. Let v_i be one of the variables in the expression $e(v_0, \ldots, v_n)$, we then have

$$\exists v_i\,.\, e(v_0, \ldots, v_n) = e(v_0, \ldots, v_n)[v_i/\mathit{False}] \vee e(v_0, \ldots, v_n)[v_i/\mathit{True}].$$

[11] Readers interested in studying the structure of BDD graphs representing sets of states and transition relations are referred to the work by Edelkamp and Reffel [16,21]. In this article, we consider BDDs an abstract data type for manipulating Boolean functions and focus on explaining how implicit search can be performed by manipulating these functions.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

```
function FORWARD BREADTH-FIRST SEARCH(s_0(v⃗))
1   reached ← ∅; forwardFrontier_0 ← s_0; i ← 0
2   while forwardFrontier_i ∧ G = ∅
3     i ← i + 1
4     forwardFrontier_i ← IMG(forwardFrontier_{i−1}) ∧ ¬reached
5     reached ← reached ∨ forwardFrontier_i
6     if forwardFrontier_i = False return failure
7   return EXTRACTSOLUTION(forwardFrontier)
```

Fig. 9. BDD-based forward breadth-first search.

Existentially quantifying a Boolean variable vector involves quantifying each variable in turn.

To illustrate the image computation, consider the first step of a search from s_0 in the example problem. We have $S(v_0, v_1) = \neg v_0 \wedge \neg v_1$. Thus

$$\begin{aligned}\text{IMG}(S) &= \big(\exists(v_0, v_1)\,.\,\neg v_0 \wedge \neg v_1 \wedge T(v_0, v_1, v_0', v_1')\big)\big[(v_0', v_1')/(v_0, v_1)\big] \\ &= (v_0' \wedge \neg v_1' \vee \neg v_0' \wedge v_1')\big[(v_0', v_1')/(v_0, v_1)\big] \\ &= v_0 \wedge \neg v_1 \vee \neg v_0 \wedge v_1,\end{aligned}$$

which as expected corresponds to state $(1, 0)$ and $(0, 1)$. It is straightforward to implement uninformed or blind BDD-based search algorithms using the image and preimage computations. The forward breadth-first search algorithm, shown in Fig. 9, computes the set of frontier states with the image computation. The set *reached* contains all explored states and is used to prune a new frontier from previously visited states. A solution is constructed by traversing the forward frontiers backward from a reached goal state to the initial state. This computation always has much lower complexity than the forward search, since the preimage computation in each iteration can be restricted to a BDD representing a single state.

Backward breadth-first search can be implemented in a similar fashion using the preimage to find the frontier states. The two algorithms are easily combined into a bidirectional search algorithm. In each iteration, this algorithm either computes the frontier states in forward or backward direction. If the set of frontier states is empty the algorithm returns failure. If an overlap between the frontier states and the reached states in the opposite direction is found the algorithm extracts and returns a solution. Otherwise the search continues. A good heuristic for deciding which direction to search in is simply to choose the direction where the previous frontier took least time to compute. When using this heuristic, bidirectional search has similar or better performance than both forward and backward search, since it will transform into one of these algorithms if the frontiers always are faster to compute in a particular direction.[12]

6.3. Partitioning

A common problem when computing the image and preimage is that the intermediate BDDs tend to be large compared to the BDD representing the result. Another problem is that the transition relation may grow very large if represented by a single BDD (a *monolithic* transition relation). In symbolic model checking one of the most successful approaches to solve these problems is *transition relation partitioning* [11]. The technique relies on the observation that a system often can be characterized as either *asynchronous* with interleaved activity or *synchronous* with simultaneous activity. Consider the system model shown in Fig. 10. During each transition of the system, the state variables V are updated. Assume that subsystem i determines the next value of the state variables Y_i' given the current value of the state variables X_i and is characterized by the transition relation $P_i(\vec{x}_i, \vec{y}\,'_i)$. If the system is asynchronous, only a single of the m subsystems is active during a transition and only the next state variables of this subsystem change value. Otherwise, if the system is synchronous, each subsystem is active during a transition. In the asynchronous case, the total transition relation is given by

$$T(\vec{v}, \vec{v}) = \bigvee_{i=1}^{m}\left(P_i(\vec{x}_i, \vec{y}_i') \wedge \bigwedge_{v' \notin Y_i'} (v' \Leftrightarrow v)\right).$$

[12] Unless a first step in an inferior direction dominates the total search time. However, we have not experienced this in practice.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

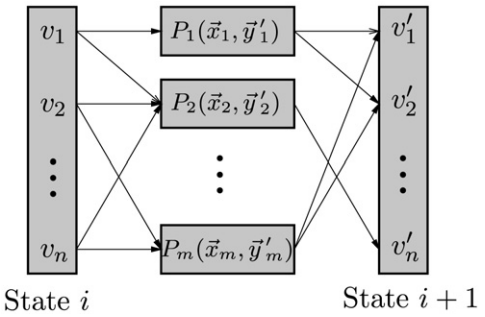

Fig. 10. System model.

To ease the presentation, assume that Y_i for $i = 1..m$ in the synchronous case is a partitioning of the state variables.[13] The transition relation in the synchronous case is then given by

$$T(\vec{v}, \vec{v}) = \bigwedge_{i=1}^{m} P_i(\vec{x}_i, \vec{y}'_i).$$

Thus, the transition relation is either represented by a *disjunctive partitioning* or a *conjunctive partitioning* of subrelations.

The main point about partitioning is that the complete transition relation never needs to be computed since both the image and preimage computations can be carried out directly on the subrelations. The asynchronous system model fits to most search problems since these often are characterized by changing a small subset of the state variables during each transition. The image computation in the asynchronous case is

$$\text{IMG}(S) = \bigvee_{i=1}^{m} \big(\exists \vec{y}_i \,.\, S(\vec{v}) \wedge P_i(\vec{x}_i, \vec{y}'_i)\big)[\vec{y}'_i/\vec{y}_i].$$

A similar approach can be used to simplify the preimage computation. Notice that we exploit that all variables except the ones modified by the active subsystem are unchanged. Thus, no quantification over these variables is necessary. This often has a substantial positive effect on the complexity of the computation. The reason is that the complexity of quantification on BDDs may be exponential in the number of quantified variables. In practice, it is often an advantage to merge some of the subrelations [50] and combine the quantification and disjunction operation to a single specialized BDD operation.

For the domain shown in Fig. 8, we can merge the transitions into two partitions P_1 and P_2 of a disjunctive partitioning, where P_1 only modifies v_0 and P_2 only modifies v_1. P_1 consists of transitions $(0,0) \to (1,0)$, $(0,1) \to (1,1)$, and $(1,0) \to (0,0)$, and P_2 consists of transitions $(0,0) \to (0,1)$ and $(1,1) \to (1,0)$

$$P_1\big((v_0, v_1), (v'_0)\big) = \neg v_0 \wedge \neg v_1 \wedge v'_0 \vee \neg v_0 \wedge v_1 \wedge v'_0 \vee v_0 \wedge \neg v_1 \wedge \neg v'_0,$$
$$P_2\big((v_0, v_1), (v'_1)\big) = \neg v_0 \wedge \neg v_1 \wedge v'_1 \vee v_0 \wedge v_1 \wedge \neg v'_1.$$

The synchronous system model fits to search problems where each transition is due to simultaneous activity (e.g., centralized multi-agent planning [34]). The image computation is more complicated in the conjunctive case due to the fact that existential quantification does not distribute over conjunction. However, a subrelation can be moved out of scope of an existential quantification if it does not depend on any of the variables being quantified. This technique is often referred to as *early quantification*. We get

$$\text{IMG}(S) = \big(\exists \vec{z}_m \,.\, \big(\cdots \big(\exists \vec{z}_1 \,.\, S(\vec{v}) \wedge P_1(\vec{x}_1, \vec{y}'_1)\big) \cdots\big) \wedge P_m(\vec{x}_m, \vec{y}'_m)\big)[\vec{v}'/\vec{v}],$$

where $Z_i \cap \bigcup_{j=i+1}^{m} X_j = \emptyset$ for $1 \leqslant i < m$ and $\bigcup_{i=1}^{m} Z_i = V$. Again, a similar approach can be used to simplify the preimage computation.

[13] It is easy extend the approaches to the general case.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

As an example, consider a system with two state variables v_0 and v_1 and two concurrent activities $P_1(v_0, v_0') = \neg v_0 \wedge v_0'$ and $P_2(v_1, v_1') = \neg v_1 \wedge v_1'$. An image computation using early quantification from the state $S(\vec{v}) = \neg v_0 \wedge \neg v_1$ would then be

$$\begin{aligned}
\text{IMG}(S) &= \big(\exists \vec{v}_1 \,.\, \big(\exists \vec{v}_0 \,.\, S(\vec{v}) \wedge P_1(v_0, v_0')\big) \wedge P_2(v_1, v_1')\big)[\vec{v}'/\vec{v}] \\
&= \big(\exists \vec{v}_1 \,.\, (\exists \vec{v}_0 \,.\, \neg v_0 \wedge \neg v_1 \wedge v_0') \wedge P_2(v_1, v_1')\big)[\vec{v}'/\vec{v}] \\
&= \big(\exists \vec{v}_1 \,.\, (\neg v_1 \wedge v_0') \wedge \neg v_1 \wedge v_1'\big)[\vec{v}'/\vec{v}] \\
&= v_0 \wedge v_1.
\end{aligned}$$

Thus as expected, the image contains a single state where the value of both state variables has been changed from *False* to *True*.

A large number of heuristics have been developed for choosing and arranging partitions in the conjunctive case (e.g., [43,50]). The main idea is to avoid a blow up of the intermediate BDDs of the image and preimage computation by reducing the life span of variables. Assume that a variable is introduced in the computation by partition i and that the variable is removed again by the existential quantification associated with partition j. The life span of the variable is then $j - i$.

6.4. The BDD-based BSFS algorithm

The BSFS algorithm represents the states in each search node by a BDD. This may lead to exponential space savings compared to the explicit state representation used by the BFS algorithm. In addition, search nodes with similar BDDs may share structure in the multi-rooted BDD representation. This may further reduce the memory consumption substantially.

However, if we want an exponential space saving to translate into an exponential time saving, we also need an implicit approach for computing the expand operation. The image computation can be applied to find all next states of a set of states implicitly, but we need a way to partition the next states into child nodes with the same cost estimates. The expand operation could be carried out in two phases, where the first finds all the next states using the image computation, and the second splits this set of states into child nodes [59]. A more direct approach, however, is to split up the image computation such that the two phases are combined into one. We call this a branching partitioning.

6.4.1. Disjunctive branching partitioning

For disjunctive partitioning the approach is straightforward. We simply ensure that each partition contains transitions with the same cost estimate change. The result is called a *disjunctive branching partitioning*.

Definition 4 *(Disjunctive branching partitioning).* A disjunctive branching partitioning is a disjunctive partitioning $P_1(\vec{x}_1, \vec{y}_1'), \ldots, P_m(\vec{x}_m, \vec{y}_m')$ where each subrelation represents a set of transitions with the same cost estimate change.

Notice, that there may exist several partitions with the same cost estimate change. This makes it possible to optimize disjunctive branching partitionings such that each partition only modifies a small set of next states variables.

So far, an unresolved problem is how to find the cost estimate change of each transition efficiently. Since cost estimates are based on a heuristic function h, this involves determining δh for each transition. It is intractable to compute $h(s)$ explicitly for each state since the number of states grows exponentially with the number of state variables of the domain. In practice, however, it turns out that δh of an action often is independent of which state it is applied in. This is not a coincidence. Heuristics are relaxations that typically are based on ignoring interactions between actions in the domain. Thus, the effect of an action can often be associated with a particular δh value. In the worst case, it may be necessary to encode the heuristic function symbolically with a BDD $h(\vec{b}, \vec{v})$ where the vector of Boolean variables $\vec{b}$ encodes the heuristic value in binary of the state represented by $\vec{v}$. We can then compute $\delta h(s, s')$ symbolically with

$$\delta h(\vec{v}, \vec{v}', \vec{d}) \equiv h(\vec{b}, \vec{v}) \wedge h(\vec{b}', \vec{v}) \wedge \vec{d} = \vec{b}' - \vec{b},$$

where $\vec{d}$ encodes the value of $\delta h(s, s')$ in binary. This computation avoids iterating over all states. In addition, it only needs to be carried out once prior to search. For all of the heuristics studied in this article (including several classical

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

```
function DISJUNCTIVESTATESETEXPAND(⟨S(v⃗), e⃗⟩)
1   child ← emptyMap
2   for i = 1 to |P|
4     e⃗_c ← e⃗ + δe⃗_i
5     child[e⃗_c] ← child[e⃗_c] ∨ IMG_i(S)
6   return MAKENODES(child)
```

Fig. 11. The state set expand function for a disjunctive branching partitioning.

heuristics), it has not been necessary to perform this symbolic computation. Instead, the δh value of each action has been independent or close to independent of the state the action is applied in.

For the domain shown in Fig. 8, a valid disjunctive branching partitioning is

$$\begin{aligned}
P_1\big((v_0, v_1), (v_0)\big) &= \neg v_0 \wedge \neg v_1 \wedge v_0' \vee \neg v_0 \wedge v_1 \wedge v_0', & \delta f_1 &= 0,\\
P_2\big((v_0, v_1), (v_0)\big) &= v_0 \wedge \neg v_1 \wedge \neg v_0', & \delta f_2 &= 2,\\
P_3\big((v_0, v_1), (v_1)\big) &= \neg v_0 \wedge \neg v_1 \wedge v_1', & \delta f_3 &= 0,\\
P_4\big((v_0, v_1), (v_1)\big) &= v_0 \wedge v_1 \wedge \neg v_1', & \delta f_4 &= 3.
\end{aligned}$$

Assume that $\mathbf{P}$ is a disjunctive branching partitioning where the cost estimate change associated with subrelation i is $\delta\vec{e}_i$. Let $\text{IMG}_i(S)$ denote the image of the transitions in subrelation i

$$\text{IMG}_i(S) \equiv \big(\exists \vec{y}_i \,.\, S(\vec{v}) \wedge P_i(\vec{x}_i, \vec{y}_i')\big)[\vec{y}_i'/\vec{y}_i].$$

The STATESETEXPAND function in Fig. 6 can then be implemented with BDDs as shown in Fig. 11. We assume that $child[\vec{e}] = \mathit{False}$ if no entry exists in *child* with key $\vec{e}$.

6.4.2. Conjunctive branching partitioning

An efficient implicit node expansion computation is also possible to define for a conjunctive partitioning. Consider the synchronous composition of the m subsystems in Fig. 10. Assume that the cost estimate change of a joint activity equals the sum of cost estimate changes of each activity. We can then represent a conjunctive branching partitioning as m disjunctive branching partitionings where each disjunctive branching partitioning represents the subrelations of the activities.

Definition 5 *(Conjunctive branching partitioning).* A conjunctive branching partitioning $P_1, \ldots, P_m$ is a set of disjunctive branching partitionings $P_i(\vec{x}_i, \vec{y}_i') = R_i^1(\vec{x}_i, \vec{y}_i'), \ldots, R_i^{r_i}(\vec{x}_i, \vec{y}_i')$ for $1 \leqslant i \leqslant m$.

Since the subsystems are synchronous, we require that the sets of variables in $\vec{y}_1', \ldots, \vec{y}_m'$ form a partitioning of the state variables V'. Assume that the cost estimate change of $R_i^j(\vec{x}_i, \vec{y}_i')$ is $\delta\vec{e}_i^{\,j}$. As an example, R_i^j could represent action transitions with cost estimate change $\delta\vec{e}_i^{\,j}$ of agent i in a multi-agent system consisting of m synchronized agents.

Further let

$$\text{SUBCOMP}_i^j(\phi) \equiv \exists \vec{z}_i \,.\, \phi(\vec{v}, \vec{v}) \wedge R_i^j(\vec{x}_i, \vec{y}_i'),$$

where ϕ represents an intermediate computation result. As for an ordinary conjunctive image computation, we require $Z_i \cap \bigcup_{j=i+1}^m X_i = \emptyset$ for $1 \leqslant i < m$ and $\bigcup_{i=1}^n Z_i = V$. The conjunctive state-set expansion function is then defined as shown in Fig. 12. The outer loop of the function performs m iterations. In iteration i, the next value of the variables $\vec{y}_i$ is computed. In the end, the map layer_i contains sets of next states of subsystem 1 to i with identical cost estimates. We assume $\mathit{layer}_i[\vec{e}] = \mathit{False}$ if no entry exists in layer_i with key $\vec{e}$. In the worst case, the number of child nodes will grow exponentially with the number of activities. However, in practice this blow-up of child nodes may be avoided due to the merging of nodes with identical cost estimates during the computation.

As an example consider computing CONJUNCTIVESTATESETEXPAND($\langle S, 5\rangle$) for some set of states S for a problem with a scalar cost estimate e and four concurrent subsystems each with transitions either changing e with -1, 0, or 1. Thus $\delta e_i^1 = -1$, $\delta e_i^2 = 0$, and $\delta e_i^3 = 1$ for $i = 1..4$. Fig. 13 shows the entries in $\mathit{layer}_0, \ldots, \mathit{layer}_4$. As depicted, the number of entries in the final layer is 9. For the kind of activities considered in this example, the number of child nodes only grows linearly with the number of concurrent activities.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

```
function ConjunctiveStateSetExpand(⟨S(v⃗), e⃗⟩)
1   layer_0 ← emptyMap
2   layer_0[e⃗] ← S
3   for i = 1 to m
4     layer_i ← emptyMap
5     foreach entry ⟨φ, e⃗_{i−1}⟩ in layer_{i−1}
6       for j = 1 to r_i
7         e⃗_i ← e⃗_{i−1} + δe⃗_i^j
8         layer_i[e⃗_i] ← layer_i[e⃗_i] ∨ SubComp_i^j(φ)
9   child ← layer_m[v⃗′/v⃗]
10  return MakeNodes(child)
```

Fig. 12. The StateSetExpand function for a conjunctive branching partitioning.

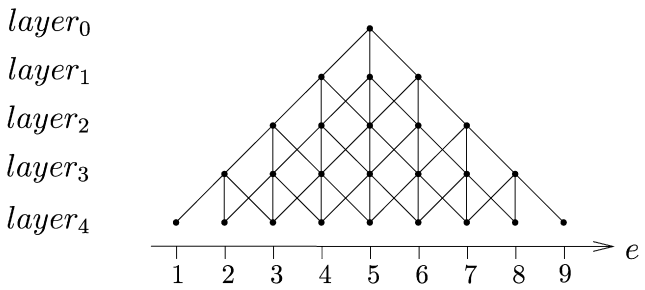

Fig. 13. Entries in $layer_i$ of ConjunctiveStateSetExpand($\langle S, 5\rangle$) for a problem with four concurrent subsystems each with transitions associated with cost estimate changes $\{-1, 0, 1\}$.

Table 1
The six search algorithms compared in the experimental evaluation

GHSETA*	The GHSETA* algorithm with evaluation function $f(n) = g(n) + h(n)$.
FSETA*	The FSETA* algorithm with evaluation function $f(n) = g(n) + h(n)$.
BIDIR	BDD-based blind breadth-first bidirectional search using the heuristic for choosing search direction described in Section 6.2.
A*	Ordinary A* with duplicate elimination, explicit state representation, and evaluation function $f(n) = g(n) + h(n)$.[15]
BDDA*	The BDDA* algorithm as described in [21].
iBDDA*	An improved version of BDDA* described below.

7. Experimental evaluation

Even though weighted A* and greedy best-first search are subsumed by the state-set branching framework, the experimental evaluation in this article focuses on algorithms performing search similar to A*. There are several reasons for this. First, we are interested in finding optimal or near optimal solutions, and for greedy best-first search, the whole emphasis would be on the quality of the heuristic function rather than the efficiency of the search approach. Second, the behavior of A* has been extensively studied, and finally, we compare with BDDA*. Readers interested in the performance of state-set branching algorithms of weighted A* with other weight settings than $w = 0.5$ (see Eq. (1)) are referred to the work by Jensen et al. [30].

We have implemented a general search engine in C++ using the BuDDy BDD package[14] [38]. This package has two major parameters: (1) the number of BDD-nodes allocated to represent the shared BDD (n), and (2) the number of BDD nodes allocated to represent BDDs in the operator caches used to implement dynamic programming (c). The input to the search engine is a search problem defined in the STRIPS part of PDDL [41] or an extended version of *NADL* [34] with action costs. The output of the search engine is a solution found by one of the six search algorithms described in Table 1.

[14] We also made experiments using the CUDD package [55], but did not obtain significantly better results than with the BuDDy package.

[15] For planning problems each state is represented by the set of true facts in the state. Since a set of states for a fixed number of facts uses space linear in the size of the set, we consider it an explicit state representation.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

function BDDA*($s_0(\vec{v})$)
1 $open(\vec{f}, \vec{v}) \leftarrow h(\vec{f}, \vec{v}) \wedge s_0(\vec{v})$
2 **while** ($open \neq \emptyset$)
3 $(f_{\min}, \min(\vec{v}), open'(\vec{f}, \vec{v})) \leftarrow$ GOLEFT($open$)
4 **if** $(\exists \vec{v} \,.\, (\min(\vec{v}) \wedge \mathcal{G}(\vec{v}))$ **return** $f_{\min}$
5 $open''(\vec{f}', \vec{v}) \leftarrow \exists \vec{v} \,.\, \min(\vec{v}) \wedge T(\vec{v}, \vec{v}) \wedge$
6 $\exists \vec{e} \,.\, h(\vec{e}, \vec{v}) \wedge \exists \vec{e}' \,.\, h(\vec{e}', \vec{v}) \wedge (\vec{f}' = f_{\min} + \vec{e}' - \vec{e} + 1)$
7 $open(\vec{f}, \vec{v}) \leftarrow open'(\vec{f}, \vec{v}) \vee open''(\vec{f}', \vec{v})[\vec{f}' \setminus \vec{f}, \vec{v}' \setminus \vec{v}]$

Fig. 14. The BDDA* algorithm.

Table 2
The performance parameters of the search engine

t_{total}	The total elapsed CPU time of the search engine.
t_{rel}	Time to generate the transition relation. For BDDA* and iBDDA*, this also includes building the symbolic representation of the heuristic function and f-formulas.
t_{search}	Time to search for and extract a solution.
$\|sol\|$	Solution length.
$\|expand\|$	For BIDIR this is the average size of the BDDs representing the search frontier. For FSETA* and GHSETA*, it is the average size of BDDs of search nodes being expanded. For BDDA* and iBDDA*, it is the average size of $open''$.
$\|Q\|_{\max}$	Maximal number of nodes on the frontier queue.
$\|T\|$	Sum of number of nodes of BDDs representing the partitioned transition relation.
it	Number of iterations of the algorithm.

The GHSETA*, FSETA*, and BIDIR search algorithms have been implemented as described in this article. The ordinary A* algorithm manipulates and represents states explicitly. For $\mathbf{FG}^k$, $D^x V^y M^z$, and the $(N^2 - 1)$-puzzles, specialized algorithms with customized state representations have been developed to minimize the space consumption. For planning problems states have been encoded explicitly as sets of facts and actions have been represented in the usual STRIPS fashion. All of the ordinary A* algorithms use the same strategy as GHSETA* to eliminate duplicates. Thus, all states that already have been visited previously with lower or equal g-costs are eliminated. The BDDA* algorithm has been implemented as described in [21]. The algorithm presented in this article is shown in Fig. 14. It can only solve search problems in domains with unit transition costs. The search frontier is represented by a single BDD $open(\vec{f}, \vec{v})$. This BDD is the characteristic function of a set of states paired with their f-cost. The state is encoded as usual by a Boolean vector $\vec{v}$ and the f-cost is encoded in binary by the Boolean vector $\vec{f}$. Similar to FSETA*, BDDA* expands all states $\min(\vec{v})$ with minimum f-cost ($f_{\min}$) in each iteration. The f-cost of the child states is computed by arithmetic operations at the BDD level (lines 5 and 6). The change in h-cost is found by applying a symbolic encoding of the heuristic function to the child and parent state. BDDA* is able to find optimal solutions, but the algorithm only returns the path cost of such solutions. In our implementation, we therefore added a function for tracing a solution backward. In the domains we have investigated, this extraction function has low complexity, as did those for GHSETA* and FSETA*. Our implementation of BDDA* shows that it often can be improved by: (1) defining a computation of $open''$ using a disjunctive partitioned transition relation instead of monolithic transition relation as in lines 5 and 6, (2) precomputing the arithmetic operation at the end of line 6 for each possible f-cost, (3) interleaving the BDD variables of $\vec{f}$, $\vec{e}$, and $\vec{e}'$ to improve the arithmetic BDD operations, and (4) moving this block of variables to the middle of the BDD variable ordering to reduce the average distance to dependent state variables. All of these improvements except the last have been considered to some degree in later versions of BDDA* [16]. The last improvement, however, is actually antagonistic to the recommendation of the BDDA* inventors who locate the $\vec{f}$ variables at the beginning of the variable ordering to simplify the GOLEFT operation. However, we get up to a factor of two speed up with the above modification. The improved algorithm is called iBDDA*.

In order to factor out differences due to state encodings and BDD computations, all BDD-based algorithms use the same bit vector representation of states, the same variable ordering of the state variables, and similar space allocation and cache sizes of the BDD package. This is necessary since a dissimilarity in just one of the above mentioned properties may cause an exponential performance difference. All algorithms share as many subcomputations as possible, but redundant or unnecessary computations are never carried out for a particular instantiation of an algorithm. The performance parameters of the search engine are shown in Table 2. Time is measured in seconds. The time $t_{\text{total}} - t_{\text{rel}} - t_{\text{search}}$ is spent on allocating memory for the BDD package, parsing the problem description and in case of PDDL problems

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

analyzing the problem in order to make a compact Boolean state encoding (the domain analysis method is explained in Section 7.4). For all domains, the size of the state space is given as the number of possible assignments to the Boolean state variables used to represent the domain. All experiments except the ones on the Pipes World and Free Cell domain are carried out on a Linux 2.4 PC with a 500 MHz Pentium III CPU, 512 KB L2 cache, and 512 MB RAM. Experiments on the Pipes World and Free Cell domain has been carried out on a Linux 2.6 PC with a 3.20 GHz Intel Xeon CPU, 1024 KB L2 cache, and 3 GB RAM. Time out and out of memory are indicated by *Time* and *Mem*. Time out changes between the experiments. The algorithms are considered out of memory when they start page faulting to the hard drive.

Our experiments cover a wide range of search domains and heuristics. The first domain FG^k uses the minimum Hamming distance as heuristic function. It has been artificially designed to demonstrate that GHSETA* may have exponentially better performance than single-state A*. Next, we consider another artificial domain called the $D^x V^y M^z$ puzzle again using the minimum Hamming distance as heuristic function. The purpose of this domain is to show the scalability of state-set branching as a function of the dependency between objects in the domain. In particular, it demonstrates how the u parameter of GHSETA* can be used to focus the search on a subset of optimal paths when there is an abundance of these. We then turn to studying several well-known search domains including the (N^2-1)-puzzles and STRIPS [22] planning problems from the international planning competitions 1998–2004. We start by examining the 24- and 35-puzzles using the usual sum of Manhattan distances as heuristic function. The planning domains include Blocks World, Logistics, Gripper, Zeno Travel, Pipes World, and Free Cell. The experiments on planning domains are interesting since they consider a backward search guided by an approximation to the HSPr heuristic [8]. In the final experiment, we show an example of state-set branching using a conjunctive branching partitioning. We study a range of channel routing problems from VLSI design produced from two circuits of the ISCAS-85 benchmarks [57] using a specialized heuristic function.

7.1. $\mathbf{FG}^k$

This problem is a modification of Barret and Weld's D^1S^1 problem [4]. The problem is easiest to describe in STRIPS. Thus, a state is a set of facts and actions are fact triples defining sets of transitions. In a given state S, an action defined by $\langle pre, add, del\rangle$ is applicable if $pre \subseteq S$, and the resulting state is $S' = (S \cup add) \setminus del$. The actions are

$\mathbf{A}_1^1$	$\mathbf{A}_i^1, \quad i = 2,\ldots,n$	$\mathbf{A}_i^2, \quad i = 1,\ldots,n$
$pre : \{F^*\}$	$pre : \{F^*, G_{i-1}\}$	$pre : \{\}$
$add : \{G_1\}$	$add : \{G_i\}$	$add : \{F_i\}$
$del : \{\}$	$del : \{\}$	$del : \{F^*\}$.

Each action is assumed to have unit cost. The initial state is $\{F^*\}$ and the goal state is $\{G_i \mid k < i \leqslant n\}$. Action $\mathbf{A}_i^1$ produces G_i given that G_{i-1} and F^* belong to the current state. In each state, however, the actions $\mathbf{A}_1^2,\ldots,\mathbf{A}_n^2$ are also applicable and they consume F^*. Thus, if one of these actions is applied no further $\mathbf{A}_i^1$ actions can be applied. This means that the only solution is $\mathbf{A}_1^1,\ldots,\mathbf{A}_n^1$. The purpose of the $\mathbf{A}_i^2$ actions is to make the decision of which action to apply in each state non-trivial. Without guidance the average number of states that must be visited in order to find a solution grows exponentially with the search depth.

This domain has been artificially designed to demonstrate the advantage of using BDDs to implicitly represent sets of states as done by GHSETA* compared to representing states explicitly as done by the ordinary single-state A* algorithm.

A state is represented by a vector of Boolean state variables

$$(G_1,\ldots,G_n,F_1,\ldots,F_n,F^*).$$

Hence, in the initial state F^* is true, while all the other state variables are false. In a goal state, the state variables $G_{k+1},\ldots,G_n$ are true while all other state variables may have arbitrary truth value. The heuristic value $h(s)$ of a state s is the minimum Hamming distance to a goal states. That is the number of goal state variables $(G_{k+1},\ldots,G_n)$ that are false in the state s. Since the heuristic function gives no information to guide the search on the first k steps, we may expect the complexity of the ordinary A* algorithm to grow exponentially with k.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

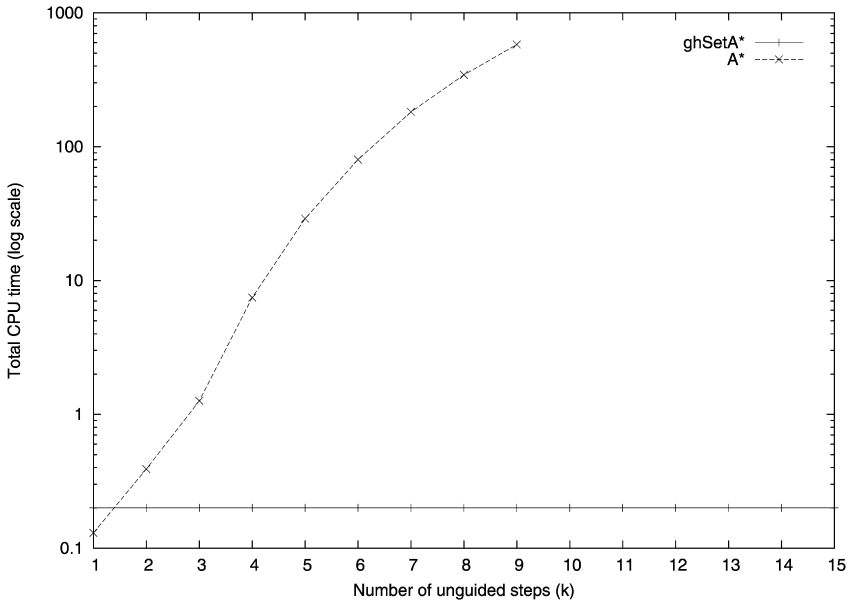

Fig. 15. Total CPU time of the FG^k problems.

In this experiment, we only compare the total CPU time and number of iterations of GHSETA* and single-state A*. The FG^k problems are defined in *NADL*. A specialized poly-time BDD operation for splitting *NADL* actions into transitions with the same cost estimate change is used for GHSETA* in the preprocessing phase. No upper bound ($u = \infty$) is used by GHSETA* and no upper limit of the branching partitions is applied. For the FG^k problems considered, n equals 16. This corresponds to 33 bits in the BDD encoding of the domain. The parameters of the BDD package are hand tuned in each experiment for best performance. Time out is 600 seconds. The results are shown in Fig. 15. The performance of A* degrades quickly with the number of unguided steps. A* gets lost expanding an exponentially growing set of states. The GHSETA* algorithm is hardly affected by the lack of guidance. An analysis of the unguided frontier layers shows that these form expressions that can be represented by symmetric functions. Since these functions can be represented by polynomial sized BDDs, GHSETA* is able to perform an ordinary BDD-based blind forward search on the unguided frontier layers using only polynomial time. Thus, the performance difference between A* and GHSETA* grows exponentially.

7.2. $D^x V^y M^z$

The $D^x V^y M^z$ domain is an artificial puzzle domain where the dependency between objects in the domain can be adjusted without changing the number of bits in the state encoding. The domain has the minimum Hamming distance as an admissible heuristic. It consists of a set of sliding tokens that can be moved between the corner positions of hypercubes. In any state, a corner position can be occupied by at most one token. Each action moves a single token to an empty adjacent corner. The dimension of the hypercubes is y. That is, the hypercubes are described by y Boolean variables. For $y = 3$ the hypercubes are regular three dimensional cubes with 8 corners. Each corner is associated with a particular assignment of the y Boolean variables. We enumerate the corners according to the value encoded in binary of the Boolean variables. Hence, an action simply flips the value of one of these Boolean variables. There are z tokens of which x are moving on the same hypercube. The remaining $z - x$ tokens are moving on individual hypercubes. This means that there is a total of $z - x + 1$ hypercubes. Tokens on individual hypercubes do not interact with other tokens. Thus, the x parameter can be used to adjust the dependency between tokens without changing the number of bits in the state encoding.

The tokens are numbered. Initially, each token is located at a corner position with the same number. There are 2^y corners on each hypercube. The goal is to move a token with number n to the corner with number $2^y - n - 1$. Each action is assumed to have unit cost. Fig. 16 shows the initial state of $D^3 V^3 M^6$.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

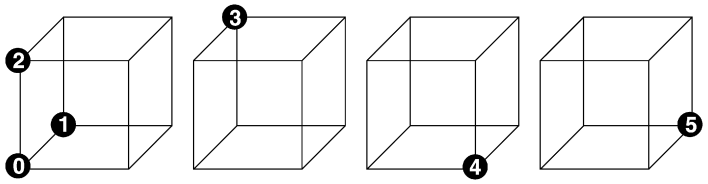

Fig. 16. The initial state of $D^3V^3M^6$.

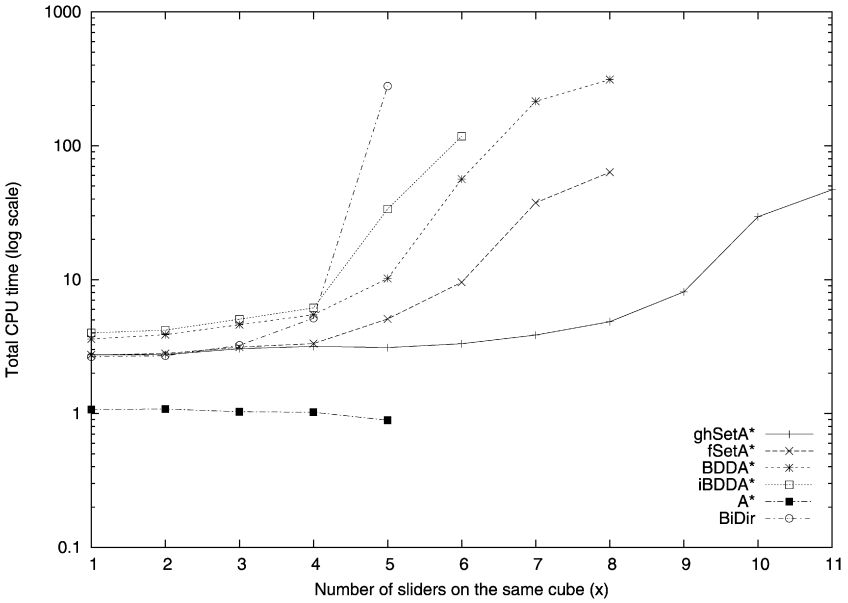

Fig. 17. Total CPU time of the $D^xV^4M^{15}$ problems.

When $x = z$ all tokens are moving on the same cube. If further $x = 2^y - 1$ all corners of the cube except one will be occupied making it a permutation problem similar to the 8-puzzle. The key idea about this problem is that the x parameter allows the dependency of tokens to be adjusted linearly without changing the number of bits used to encode a state. In addition, it demonstrates how the u parameter of the GHSETA* algorithm can be used to focus the search when there is an abundance of optimal paths to explore. For the BDD-based algorithms, the $D^xV^4M^{15}$ problems are defined in *NADL*. Again a specialized poly-time BDD operation for splitting *NADL* actions into transitions with the same cost estimate change is applied by GHSETA* and FSETA*. For all problems, the number of bits in the BDD encoding is 60. For GHSETA* the upper bound for node merging is 200 ($u = 200$). All BDD-based algorithms except BDDA* utilize a disjunctive partitioning with an upper bound on the BDDs representing a partition of 5000. Time out is 500 seconds. For all problems, the BDD-based algorithms use 2.3 seconds on initializing the BDD package ($n = 8000000$ and $c = 700000$).[16] The results are shown in Table 3. Fig. 17 shows a graph of the total CPU time for the algorithms.

All solutions found are 34 steps long. For BDDA* and iBDDA* the size of the BDD representing the heuristic function is 2014 and 1235, respectively. Both the size of the monolithic and partitioned transition relation grows fast with the dependency between tokens. The problem is that there is no efficient way to model whether a position is occupied or not. The most efficient algorithm is GHSETA*. The FSETA* algorithm has worse performance than GHSETA* because it has to expand all states with minimum f-cost in each iteration, whereas GHSETA* focus on a

[16] Notice that we choose to allocate a large number of nodes even for the small problems. The reason is that we mainly care about the asymptotic performance of the algorithms. Better results can be obtained on the small problems by adjusting the number of nodes to the size of the problem (e.g., by doubling an initially small number of nodes every time the BDD package runs out of free nodes).

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Table 3
Results of the $D^x V^4 M^{15}$ problems

Algorithm	x	t_{total}	t_{rel}	t_{search}	$\lvert expand \rvert$	$\lvert Q \rvert_{\max}$	$\lvert T \rvert$	it
GHSETA*	1	2.7	0.3	0.2	307.3	33	710	34
	2	2.8	0.3	0.2	307.3	33	1472	34
	3	3.1	0.4	0.3	671.0	33	4070	34
	4	3.2	0.5	0.4	441.7	72	10292	34
	5	3.1	0.4	0.4	194.8	120	20974	34
	6	3.3	0.6	0.4	139.9	212	45978	34
	7	3.9	1.0	0.5	128.4	322	104358	34
	8	4.9	1.9	0.6	115.9	438	232278	34
	9	8.1	5.0	0.8	132.0	557	705956	34
	10	29.5	14.3	12.8	146.1	5103	1970406	373
	11	46.9	43.8	0.8	107.3	336	5537402	34
	12	*Mem*						
FSETA*	1	2.7	0.3	0.2	307.3	1	710	34
	2	2.8	0.3	0.2	307.3	1	1472	34
	3	3.1	0.4	0.4	671.0	1	4070	34
	4	3.3	0.4	0.6	671.0	1	10292	34
	5	5.1	0.5	2.3	1778.6	1	20974	34
	6	9.6	0.6	6.6	2976.5	1	45978	34
	7	37.5	1.0	34.2	9046.7	1	104358	34
	8	63.4	2.0	59.1	9046.7	1	232278	34
	9	408.3	4.9	401.1	24175.4	1	705956	34
	10	*Time*						
BDDA*	1	3.6	0.5	0.4	314.3		355	34
	2	3.9	0.5	0.6	314.3		772	34
	3	4.6	0.6	1.3	678.0		2128	34
	4	5.5	0.8	2.0	678.0		6484	34
	5	10.2	1.3	6.2	1785.6		20050	34
	6	56.4	3.4	50.4	2983.5		64959	34
	7	214.8	10.8	201.1	9053.7		234757	34
	8	312.1	52.7	256.1	9053.7		998346	34
	9	*Time*						
iBDDA*	1	4.0	0.4	0.8	307.3		355	34
	2	4.2	0.4	1.1	307.3		772	34
	3	5.1	0.5	1.9	671.0		2128	34
	4	6.2	0.4	3.0	671.0		6791	34
	5	33.7	0.4	30.4	1778.6		25298	34
	6	117.6	0.5	113.9	2976.5		84559	34
	7	*Time*						
A*	1	1.1				1884		34
	2	1.1				1882		34
	3	1.0				1770		34
	4	1.0				1750		34
	5	0.9				1626		34
	6	*Time*						
BIDIR	1	2.7	0.2	0.1	568.5		355	34
	2	2.7	0.2	0.2	630.8		772	34
	3	3.2	0.3	0.7	2305.1		2128	34
	4	5.2	0.2	2.6	3131.1		5159	34
	5	278.9	0.2	276.4	30445.0		10610	34
	6	*Time*						

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	

Fig. 18. Goal state of the 24-puzzle.

subset of them by having $u = 200$. A subexperiment shows that GHSETA* has similar performance to FSETA* when setting $u = \infty$. The impact of the u parameter is significant for this problem since, even for fairly large values of x, it has an abundance of optimal solutions. As can be seen for problem 10, however, the low value of u may also lead to more search. BDDA* has much worse performance than FSETA* even though it expands the exact same set of states in each iteration. As we show in Section 7.6, the problem is that the complexity of the computation of $open''$ grows fast with the size of the BDD representing the states to expand. Surprisingly the performance of iBDDA* is worse than BDDA*. This is unusual, as the remaining experiments will show. The reason might be that only a little space is saved by partitioning the transition relation in this domain. This may cause the computation of $open''$ for iBDDA* to deteriorate because it must iterate through all the partitions. A* performs well when $f(n)$ is a perfect or near perfect discriminator, but it soon gets lost in keeping track of the fast growing number of states on optimal paths. It times out in a single step going from about one second to more than 500 seconds. The problem for BIDIR is the usual for blind BDD-based search algorithms applied to hard combinatorial problems: the BDDs representing the search frontiers blow up.

7.3. The 24- and 35-puzzle

We have further analyzed "non-artificial" domains. We aim at using domains that embed a search with a potential significant large number of search states. We turned to investigating the $(N^2 - 1)$-puzzles, in particular the 24-puzzle ($n = 5$) and the 35-puzzle ($n = 6$). The domain consists of an $n \times n$ board with $n^2 - 1$ numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The task is to reach the goal configuration as shown for the 24-puzzle in Fig. 18. For our experiments, the initial state is generated by performing r random moves from the goal state.[17] We assume unit cost transitions and use the well-known sum of Manhattan distances of the tiles to their goal position as heuristic function. This heuristic function is admissible. For GHSETA* and FSETA* a disjunctive branching partitioning is easy to compute since δh of an action changing the position of a single tile is independent of the position of the other tiles. The two algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For the BDD-based algorithms, the problems are defined in *NADL* and the best results are obtained when having no limit on the partition size. Thus, BDDA*, iBDDA*, and BIDIR use a monolithic transition relation. The number bits in the BDD encoding of the 24-puzzle is 125. The results of this experiment are shown in Table 4. For all 24-puzzle problems, the BDD-based algorithms spend 3.6 seconds on initializing the BDD package ($n = 15000000$ and $c = 500000$). Time out is 10000 seconds. For BDDA* and iBDDA* the size of the BDD representing the heuristic function is 33522 and 18424, respectively. For GHSETA* and FSETA* the size of the transition relations is 70582, while the size of the transition relation for BDDA* and iBDDA* is 66673. Thus a small amount of space was saved by using a monolithic transition relation representation. However, GHSETA* and FSETA* have better performance than BDDA* and iBDDA* mostly due to the their more efficient node expansion computation. Interestingly, both BDDA* and iBDDA* spend significant time computing the heuristic function in this domain. The GHSETA* and FSETA*

[17] In each of these steps choosing the move back to the previous state is illegal.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Table 4
Results of the 24-puzzle problems

Algorithm	r	t_{total}	t_{rel}	t_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{\max}$	it
GHSETA*	140	28.8	22.1	2.7	26	187.5	23	93
	160	30.0	22.2	3.8	28	213.2	24	175
	180	31.4	22.2	5.3	32	270.2	28	253
	200	43.7	21.9	14.9	36	786.2	31	575
	220	36.3	22.2	10.1	36	411.1	31	490
	240	199.3	22.0	173.2	50	2055.5	44	1543
	260	5673.7	23.9	5644.5	56	10641.2	48	2576
	280	*Mem*						
	300	4772.7	20.9	4743.97	60	9761.3	53	2705
	320	*Mem*						
FSETA*	140	29.7	21.0	4.7	26	669.9	1	42
	160	32.2	20.9	7.4	28	1051.6	1	57
	180	34.3	21.0	9.5	32	1207.0	1	69
	200	50.1	21.0	25.3	36	5276.0	1	93
	220	41.8	21.0	17.0	36	3117.6	1	88
	240	205.2	21.0	180.5	50	18243.3	1	156
	260	*Mem*						
BDDA*	140	98.5	83.0	11.3	26	676.9		42
	160	114.7	83.2	27.4	28	1058.6		57
	180	129.8	82.9	42.7	32	1214.0		69
	200	425.0	83.1	337.1	36	5283.0		93
	220	267.7	82.8	180.6	36	3124.6		88
	240	4120.1	83.1	4032.8	50	18250.3		156
	260	*Time*						
iBDDA*	140	79.8	66.7	5.9	26	669.9		42
	160	85.3	65.7	11.8	28	1051.6		57
	180	93.6	65.7	20.0	32	1207.0		69
	200	314.6	65.8	240.9	36	5276.0		93
	220	156.9	65.6	83.5	36	3117.6		88
	240	2150.3	65.9	2076.6	50	18243.3		156
	260	*Mem*						
A*	140	0.1			26		300	221
	160	0.9			28		725	546
	180	0.6			32		1470	1106
	200	7.4			36		15927	12539
	220	2.3			36		5228	4147
	240	87.1			50		159231	133418
	260	*Mem*						
BIDIR	140	68.1	36.6	27.9	26	34365.2		26
	160	96.0	36.8	55.6	28	55388.4		28
	180	214.7	36.8	174.3	32	106166.0		32
	200	1286.0	36.8	1245.6	36	359488.0		36
	220	3168.8	36.8	3128.4	36	421307.0		36
	240	*Mem*						

algorithms also scale better than A* and BIDIR. A* has good performance because it does not have the substantial overhead of computing the transition relation and finding actions to apply. However, due to the explicit representation of states, it runs out of memory for solution depths above approximately 50. For BIDIR, the problem is the usual: the BDDs representing the search frontiers blow up. Fig. 19 shows a graph of the total CPU time of the 24- and 35-puzzle. Again time out is 10000 seconds.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

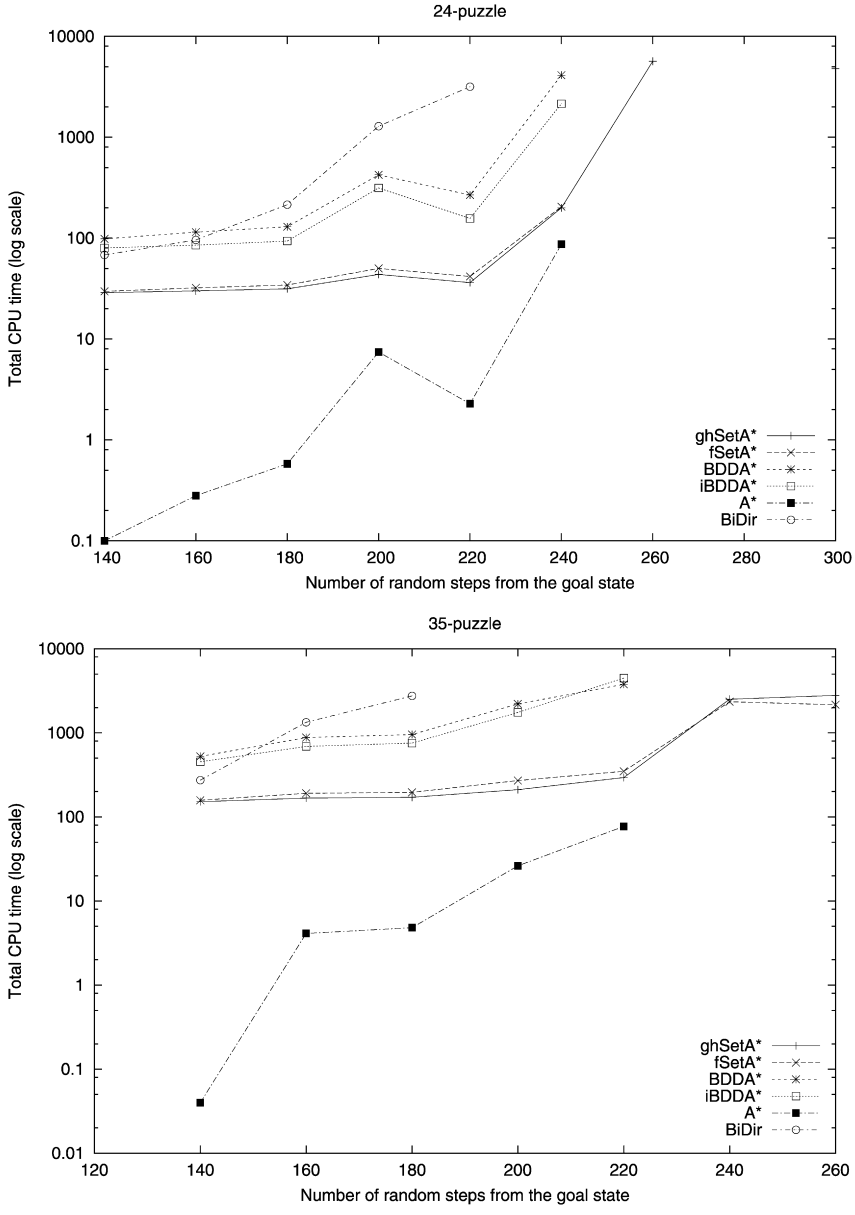

Fig. 19. Total CPU time for the 24- and 35-puzzle problems.

7.4. Planning domains

In this section, we consider six planning problems from the STRIPS track of the international planning competitions 1998–2004. The problems are defined in the STRIPS part of PDDL. An optimal solution is a solution with minimal length, so we assume unit cost actions. A Boolean representation of a STRIPS domain is trivial if using a single Boolean state variable for each fact. This encoding, however, is normally very inefficient due to its redundant representation of static facts and facts that are mutually exclusive or unreachable. In order to generate a more compact encoding, we analyze the STRIPS problem in a three step process.

(1) Find static facts by subtracting the facts mentioned in the add and delete sets of actions from the facts in the initial state.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

(2) Approximate the set of reachable facts from the initial state by performing a relaxed reachability analysis, ignoring the delete set of the actions.
(3) Find sets of *single-valued predicates* [25] via inductive proofs on the reachable facts.

If a set of predicates are mutual exclusive when restricting a particular argument in each of them to the same object then the set of predicates is said to be single-valued. Consider for instance a domain where packages can be either inside a truck $in(P, T)$ or at locations $at(P, L)$. Then *in* and at *at* are single-valued with respect to the first argument. The reachability analysis in step 2 is implemented based an approach described in the work by Edelkamp and Helmert [20]. It is fast for the problems considered in this article (for most problems less than 0.04 seconds). The algorithm proceeds in a breadth-first manner such that each fact f can be assigned a depth $d(f)$ where it is reached. Similar to the MIPS planning system [16], we use this measure to approximate the HSPr heuristic [8]. HSPr is an efficient but non-admissible heuristic for backward search. For a state given by a set of facts S, the approximation to HSPr is given by

$$h(S) = \sum_{f \in S} d(f).$$

A branching partitioning for this heuristic is efficient to generate given that each action (pre, add, del) leading from S to $S' = (S \cup add) \setminus del$ satisfies

$$del \subseteq pre \quad \text{and} \quad add \cap pre = \emptyset.$$

These requirements are natural and satisfied by all the planning domains considered in this article. Due to the constraints, we get

$$\begin{aligned} \delta h &= h(S') - h(S) \\ &= h(add \setminus S) - h(del) \\ &= \sum_{f \in add \setminus S} d(f) - \sum_{f \in del} d(f). \end{aligned}$$

Thus, each action is partitioned in up to $2^{|add|}$ sets of transitions with different δh-cost.

The HSPr heuristic is applied in a backward search rather than a regression search.[18] This affects computing the h-cost of a goal state. Consider a planning problem with k facts $\{f_1, \ldots, f_k\}$ and goal description $\mathcal{G} = (f_1)$. In a regression search, $\mathcal{G}$ represents a single state with $h(\mathcal{G}) = d(f_1)$. In a backward search, on the other hand, $\mathcal{G}$ represents 2^{k-1} states that in principle each may have different h-cost. To avoid this problem, we have extended the goal descriptions of the planning problems so they correspond to single states. This may increase the solution length of the problems. In addition, it makes the backward exploration more similar to forward exploration in that it only considers valid states. Thus, by extending the goal description we also avoid a common deterioration of BDD-based search when applied backward due to an exploration of an unstructured space consisting of a mixture of valid and invalid states.

Since the HSPr heuristic for most states in many of the studied domains either under- or overestimates the true distance to the initial state, we have manually scaled it to be as accurate as possible. The reason for this is to give a comparison with the optimal BIDIR algorithm that is as fair as possible. If the heuristic overestimates, the A* algorithms may be fast but give poor solutions. If the heuristic underestimates, the A* algorithms may give optimal solutions but be overly slow. However, despite of these adjustments the complexity difference between suboptimal and optimal search makes a direct comparison between BIDIR and the A* algorithms impossible when using an inadmissible heuristic like HSPr.

7.4.1. Blocks world

The Blocks World is a classical planning domain. It consists of a set of cubic blocks sitting on a table. A robot arm can stack and unstack blocks from some initial configuration to a goal configuration. The problems, we consider, are

[18] Using BDDs for regression search is an interesting direction for future work.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Table 5
Results of the Blocks World problems

Algorithm	p	t_{total}	t_{rel}	t_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{\max}$	it	$\|T\|$
GHSETA*	4	2.6	0.0	0.0	6	19.5	1	6	706
	5	2.7	0.1	0.1	12	33.4	11	31	1346
	6	2.6	0.1	0.1	12	57.7	9	30	2608
	7	3.1	0.2	0.4	20	53.8	48	152	4685
	8	4.1	0.3	1.3	18	540.4	12	72	7475
	9	17.0	0.4	14.1	32	331.8	94	991	8717
	10	116.2	0.6	113.1	38	744.9	111	2309	11392
	11	133.5	0.7	130.2	32	1404.9	91	1200	16122
	12	14.8	1.0	11.2	34	410.3	120	557	18734
	13	*Time*							
	14	112.1	1.7	107.8	38	1067.8	125	1061	30707
	15	*Time*							
FSETA*	4	2.5	0.0	0.0	6	29.8	1	6	706
	5	2.7	0.1	0.1	12	68.7	4	23	1346
	6	2.7	0.1	0.1	12	126.8	2	20	2608
	7	3.2	0.2	0.5	20	121.9	8	92	4685
	8	3.9	0.3	1.1	18	1328.8	2	35	7475
	9	30.0	0.4	27.1	32	935.5	10	610	8717
	10	217.0	0.6	213.8	38	2594.4	12	1098	11392
	11	259.8	0.8	256.4	32	4756.0	9	671	16122
	12	39.2	1.0	35.7	34	817.0	13	860	18734
	13	*Time*							
	14	274.3	1.7	270.0	38	1555.1	13	1462	30707
	15	*Time*							
BDDA*	4	3.3	0.0	0.1	6	37.8		6	706
	5	3.6	0.2	0.2	12	76.7		23	1365
	6	3.6	0.2	0.2	12	134.8		20	2334
	7	4.9	0.5	1.2	20	129.9		92	4669
	8	6.0	0.5	2.2	18	1336.8		35	6959
	9	100.8	1.1	96.5	32	943.5		610	9923
	10	*Time*							
iBDDA*	4	2.7	0.0	0.0	6	29.8		6	706
	5	2.8	0.1	0.1	12	68.7		23	1365
	6	2.9	0.1	0.1	12	126.8		20	2334
	7	3.7	0.3	0.7	20	121.9		92	4669
	8	6.2	0.4	3.2	18	1328.8		35	7123
	9	113.7	0.6	110.3	32	935.5		610	10361
	10	*Time*							
A*	4	0.0		0.0	6		8	15	
	5	0.2		0.2	12		62	70	
	6	0.4		0.4	12		115	102	
	7	1.3		1.2	20		287	287	
	8	31.9		31.6	18		7787	5252	
	9	233.9		232.9	32		38221	31831	
	10	*Time*							
BIDIR	4	2.6	0.0	0.0	6	124.5		6	706
	5	2.6	0.1	0.0	12	228.3		12	1423
	6	2.7	0.1	0.1	12	438.8		12	2567
	7	3.6	0.2	0.8	20	1931.3		20	5263
	8	9.7	0.3	6.8	18	11181.8		18	8157
	9	146.8	0.4	143.9	30	75040.9		30	11443
	10	*Time*							

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

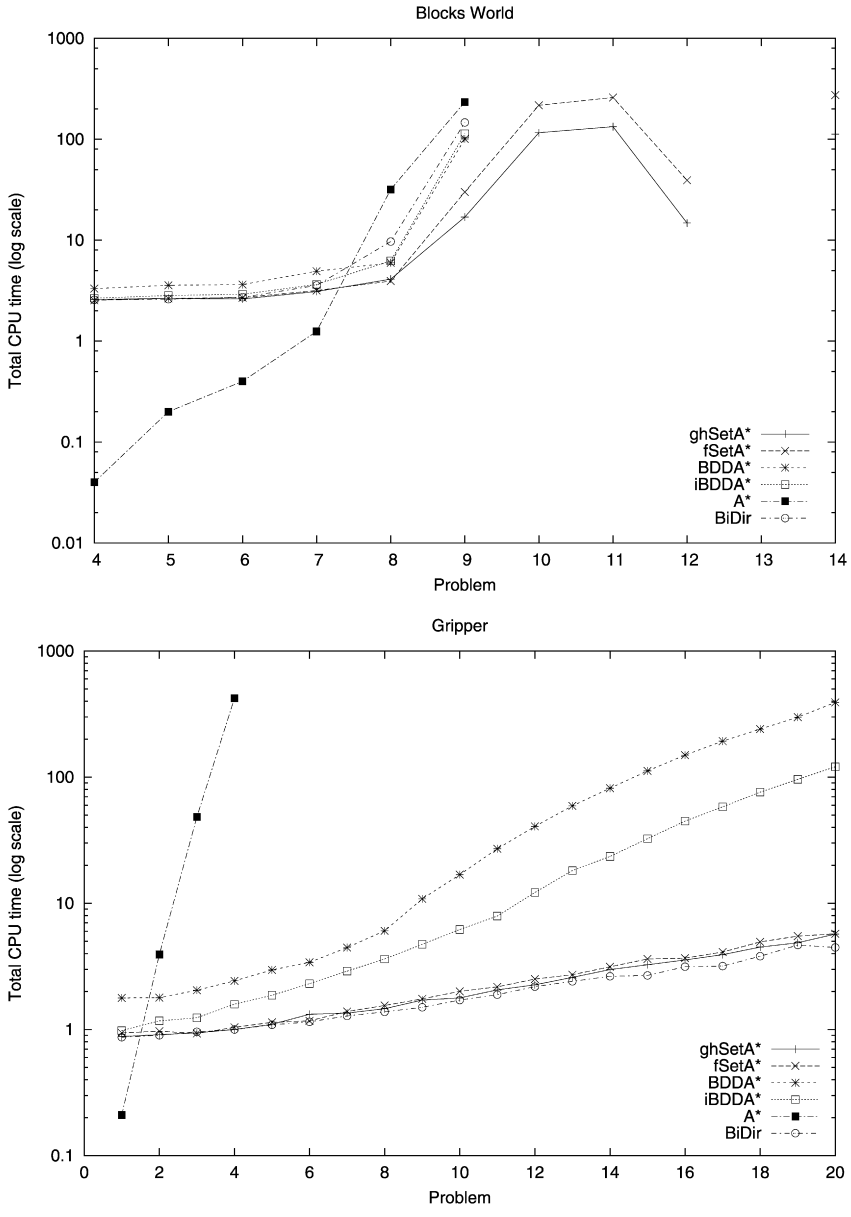

Fig. 20. Total CPU time for the Blocks World and Gripper problems.

from the untyped STRIPS track of the AIPS 2000 planning competition. The number of bits in the BDD encoding is in the range [17, 80]. The HSPr heuristic is scaled by a factor of 0.4. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs of the nodes on the frontier ($u = \infty$). For all BDD-based algorithms, the partition limit is 5000. For each problem, these algorithms spend about 2.5 seconds on initializing the BDD package ($n = 8000000$ and $c = 800000$). Time out is 500 seconds in all experiments. The results are shown in Table 5. The top graph of Fig. 20 shows the total CPU time of the algorithms. For BDDA* and iBDDA* the size of the BDD representing the heuristic function is in the range of [8, 1908] and [8, 1000], respectively. The GHSETA* and FSETA* algorithms have significantly better performance than all other algorithms. As usual BDDA* and iBDDA* suffer from an inefficient expansion computation while the frontier BDDs blow up for BIDIR. The general A* algorithm for STRIPS planning problems is less domain-tuned than the previous A* implementations. In particular, it must check

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

the precondition of all actions in each iteration in order to find the ones that are applicable. This, in addition to the explicit state representation, may explain the poor performance of A*.

7.4.2. Gripper

The Gripper problems are from the first round of the STRIPS track of the AIPS 1998 planning competition. The domain consists of a robot with two grippers which can move between two rooms connected by a door. Initially, a number of balls are located in the first room, and the goal is to move them to the other room. The number of bits in the BDD encoding is in the range [12, 88]. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms no partition limit is used, and they spend about 0.8 seconds on initializing the BDD package ($n = 2000000$ and $c = 400000$). All algorithms generate optimal solutions. The results are shown in Table 6. The bottom graph of Fig. 20 shows the total CPU time of the algorithms. Interestingly BIDIR is the fastest algorithm in this domain since the BDDs representing the search frontier only grows moderately during the search. The GHSETA* and FSETA* algorithms, however, have almost as good performance. BDDA* and iBDDA* has particularly bad performance in this domain. The problem is that the BDDs of frontier nodes grow quite large for the harder problems.

7.4.3. Logistics

The Logistics domain considers moving packages with trucks between locations in the same city and with airplanes between cities. The problems considered are from the STRIPS track of the AIPS 2000 planning competition. The number of bits in the BDD encoding is in the range [21, 86]. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms a partition limit of 5000 is used and they spend about 2.0 seconds on initializing the BDD package ($n = 8000000$ and $c = 400000$). Due to systematic under estimation, the HSPr heuristic is scaled with a factor of 1.5. The top graph of Fig. 21 shows the total CPU time of the algorithms. Only GHSETA* and FSETA* are able to solve large instances of this problem. The BDD encoding based single-valued predicates is particularly efficient in this domain. Moreover, the HSPr heuristic is quite strong which gives the A* algorithms an edge over BIDIR.

7.4.4. Zeno Travel

Zeno Travel is from the STRIPS track of the AIPS 2002 planning competition. It involves transporting people around in planes, using different modes of movement: fuel-efficient and wasteful. The number of bits in the BDD encoding is in the range [9, 165]. The GHSETA* and FSETA* algorithms have no upper bound on the size of BDDs in the frontier nodes ($u = \infty$). For all BDD-based algorithms a partition limit of 4000 is used. About 2.7 seconds are spent on initializing the BDD package ($n = 10000000$ and $c = 700000$). The bottom graph of Fig. 21 shows the total CPU time of the algorithms. The results are fairly similar to the results of the Logistics problems except that the advantage of GHSETA* and FSETA* is less significant.

7.4.5. Pipes World

The task in the Pipes World domain is to transport oil derivative products through a pipeline system. Since adding a product to a pipeline affects all the products in the pipeline, the structure of the domain is quite different from the structure of the Logistics and Zeno Travel domain. If a pipe can hold more than one product, two actions are used to model a state change of a pipeline. The first adds the product to the sender end of the pipe, while the second removes the product that is pushed out at the receiver end of the pipe. The problems, we consider, are from the typed STRIPS track of the International Planning Competition 2004. The problems have been changed manually to an untyped version. The number of bits in the BDD encoding is in the range [62, 118]. The HSPr heuristic is scaled by a factor of 0.7 due to systematic over estimation. For GHSETA* no upper bound on the size of BDDs of the nodes on the frontier is used ($u = \infty$). For all BDD-based algorithms, the partition limit is 10000 for small problems and 20000 for large problems. The number of nodes allocated by the BDD package (n) is in the range $[2M, 107M]$ and the cache size (c) is adjusted to approximately 10% of the number of nodes. Time out is 3600 seconds in all experiments. The results are shown in Table 7.[19] The top graph of Fig. 22 shows the total CPU time of the algorithms. For BDDA*

[19] The $|Q|_{\max}$ data have not been gathered for A* in the Pipes World and Free Cell domain. Moreover due to time limitations, we have not investigated the performance of FSETA* on these domains.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Table 6
Results of the Gripper problems

Algorithm	p	t_{total}	t_{rel}	t_{search}	$\|expand\|$	$\|Q\|_{\max}$	it	$\|T\|$
GHSETA*	2	0.9	0.1	0.02	68.8	5	21	594
	4	1.0	0.1	0.08	168.9	6	43	1002
	6	1.3	0.2	0.27	314.9	6	65	1410
	8	1.5	0.3	0.34	504.8	6	87	1818
	10	1.8	0.4	0.54	738.1	6	109	2226
	12	2.3	0.5	0.88	1014.7	6	131	2634
	14	3.0	0.7	1.33	1334.5	6	153	3042
	16	3.6	0.9	1.78	1697.5	6	175	3450
	18	4.5	1.1	2.46	2103.7	6	197	3858
	20	5.7	1.4	3.37	2553.1	6	219	4266
FSETA*	2	1.0	0.1	0.1	95.4	1	17	594
	4	1.0	0.1	0.1	231.2	1	29	1002
	6	1.2	0.2	0.2	423.9	1	41	1410
	8	1.6	0.3	0.3	673.4	1	53	1818
	10	2.0	0.4	0.6	979.9	1	65	2226
	12	2.5	0.6	1.0	1343.3	1	77	2634
	14	3.1	0.8	1.4	1763.5	1	89	3042
	16	3.7	0.9	1.9	2240.7	1	101	3450
	18	5.0	1.2	2.9	2774.7	1	113	3858
	20	5.7	1.5	3.2	3365.6	1	125	4266
BDDA*	2	1.8	0.1	0.2	103.4		17	323
	4	2.4	0.2	0.6	239.2		29	539
	6	3.4	0.3	1.5	431.9		41	755
	8	6.1	0.6	4.0	681.4		53	971
	10	16.9	0.9	14.4	987.9		65	1187
	12	40.7	1.2	37.9	1351.3		77	1403
	14	81.7	1.6	78.5	1771.5		89	1619
	16	149.3	2.2	145.4	2248.7		101	1835
	18	240.4	3.1	235.5	2782.7		113	2051
	20	391.1	3.9	385.5	3373.6		125	2267
iBDDA*	2	1.2	0.1	0.1	95.4		17	323
	4	1.6	0.1	0.4	231.2		29	539
	6	2.3	0.3	1.0	423.9		41	755
	8	3.6	0.4	2.2	673.4		53	971
	10	6.2	0.6	4.5	979.9		65	1187
	12	12.2	0.9	9.2	1343.3		77	1403
	14	23.5	1.1	21.3	1763.5		89	1619
	16	44.8	1.6	42.1	2240.7		101	1835
	18	76.1	2.2	72.4	2774.7		113	2051
	20	120.9	2.7	116.7	3365.6		125	2267
A*	2	3.9		3.9		698	1286	
	4	422.9		422.3		26434	85468	
	6	*Time*						
BIDIR*	2	0.9	0.1	0.0	125.4		17	323
	4	1.0	0.1	0.1	290.9		29	539
	6	1.2	0.2	0.1	589.7		41	755
	8	1.4	0.3	0.3	958.2		53	971
	10	1.7	0.4	0.5	1404.3		65	1187
	12	2.2	0.5	0.8	1611.0		77	1403
	14	2.6	0.7	1.0	2025.6		89	1619
	16	3.2	0.9	1.3	3265.6		101	1835
	18	3.8	1.2	1.7	4074.4		113	2051
	20	4.5	1.5	2.1	4944.9		125	2267

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

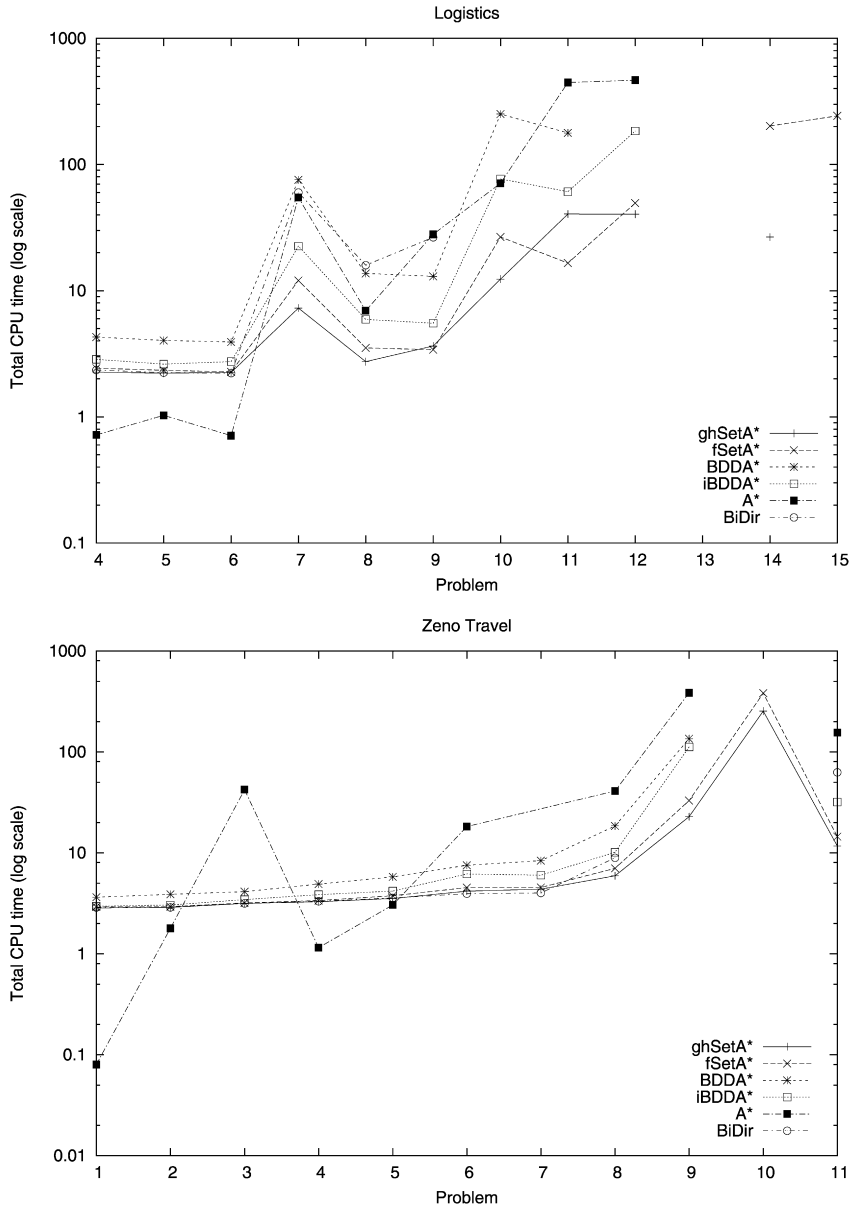

Fig. 21. Total CPU time for the Logistics and Zeno Travel problems.

and iBDDA* the size of the BDD representing the heuristic function is in the range [2244, 18577] and [988, 9743], respectively. The performance of GHSETA* and BIDIR is fairly similar while the performance of BDDA*, iBDDA*, and A* is substantially lower. Looking closer at the results of GHSETA*, A*, and BIDIR, we observe that GHSETA* and A* use considerably longer time on problem 10 compared with problem 9, while the opposite is true for BIDIR. Since BIDIR and GHSETA* represent states in the same way, the results indicate that GHSETA* traverse a larger fraction of the state space than BIDIR on problem 10. Thus, the HSPr heuristic seems to be somewhat weak on this domain sometimes guiding the exploration in the right direction (problem 9) and sometimes not (problem 10). Interestingly, the performance of A* is better than GHSETA* on problem 8. An inspection of Table 7 shows that both algorithms spend little time on search on this problem, but that GHSETA* spends considerable time constructing the BDD representation of the transition relation. Thus, this is an example of a situation where the search problem is too small for BDD-based search to pay off.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

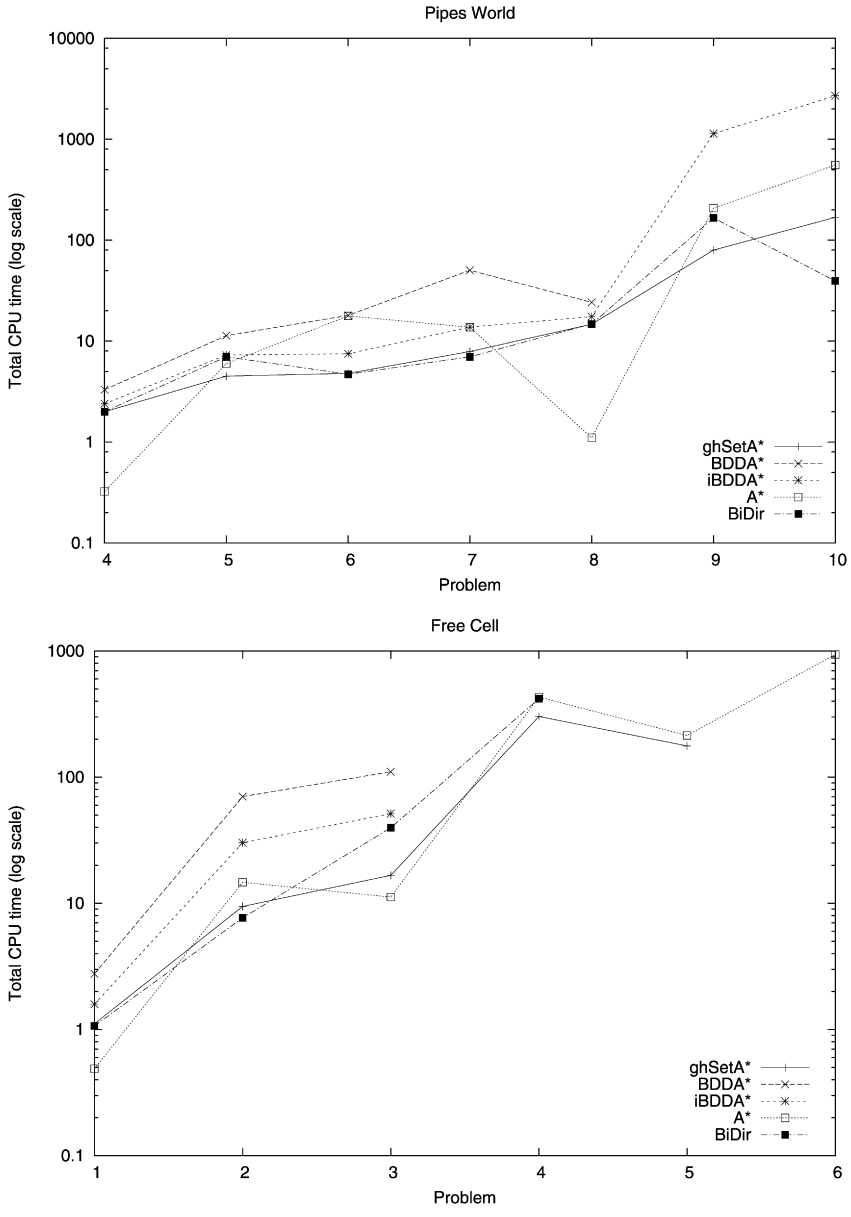

Fig. 22. Total CPU time for the Pipes World and Free Cell problems.

7.4.6. Free Cell

The Free Cell domain is a solitaire game shipped with Windows. The cards are distributed face up in 8 columns. The goal is to arrange the cards in order in four home cells. In addition to the home cells, there are four free cells. A legal action is to move a card from the bottom of a column or a free cell to a home cell holding the predecessor with matching suit, an empty free cell, an empty column, or the bottom of a column holding one of the two successors with opposite color. The problems, we consider, are from the untyped STRIPS track of the International Planning Competition 2002. The number of bits in the BDD encoding is in the range [58, 199]. The HSPr heuristic is scaled by a factor of 0.6 due to systematic over estimation. The GHSETA* algorithm has no upper bound on the size of BDDs of the nodes on the frontier ($u = \infty$). For all BDD-based algorithms, the partition limit is 10000 for small problems and 20000 for large problems. The number of nodes allocated by the BDD package (n) is in the range $[2M, 107M]$ and the cache size (c) is adjusted to approximately 10% of the number of nodes. Time out is 3600 seconds in all

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Table 7
Results of the Pipes World problems

Algorithm	p	t_{total}	t_{rel}	t_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{\max}$	it	$\|T\|$
GHSETA*	4	2.0	1.2	0.2	8	185.9	31	13	38123
	5	4.5	2.2	1.2	13	389.1	61	92	57992
	6	4.8	1.6	1.7	20	334.6	60	245	47297
	7	7.9	2.8	3.0	16	457.9	56	203	74988
	8	14.7	4.8	0.4	14	171.1	54	63	116758
	9	79.9	9.0	61.2	21	1356.3	111	680	195345
	10	168.3	3.5	161.1	30	741.2	127	4427	85022
BDDA*	4	3.3	1.7	0.5	8	204.3		14	21020
	5	11.3	3.1	6.6	13	599.3		65	31406
	6	17.9	2.9	13.2	20	414.2		457	25166
	7	50.3	5.3	42.6	16	683.6		394	40395
	8	24.2	9.0	5.4	14	453.7		54	60901
	9	*Time*							
	10	*Time*							
iBDDA*	4	2.4	1.1	0.3	8	198.1		14	39536
	5	7.3	1.9	3.4	13	593.3		65	62578
	6	7.5	1.7	3.8	20	406.2		457	37335
	7	13.8	3.1	7.5	16	675.6		394	61568
	8	17.5	4.8	1.2	14	448.8		54	80153
	9	1136.0	8.2	1112.9	21	2105.8		4384	164901
	10	2696.8	3.7	2687.8	30	902.0		52010	69929
A*	4	0.3			8			69	
	5	6.0			13			1259	
	6	17.8			20			4278	
	7	13.7			16			2651	
	8	1.1			14			139	
	9	207.7			21			20881	
	10	555.5			30			104221	
BIDIR	4	2.0	1.0	0.2	8	2646.9		8	39512
	5	7.0	1.8	3.8	13	12146.5		13	61000
	6	4.7	1.5	1.7	20	7236.4		20	37299
	7	7.0	2.6	2.2	16	10860.6		16	61001
	8	14.8	3.7	1.6	14	9841.6		14	71708
	9	166.3	6.5	149.7	21	260428.0		21	138504
	10	39.5	3.1	32.7	30	48667.4		30	68261

experiments. The results are shown in Table 8. The top graph of Fig. 22 shows the total CPU time of the algorithms. For BDDA* and iBDDA* the size of the BDD representing the heuristic function is in the range of [553, 4427] and [366, 2060], respectively. Again, we observe that BDDA* and iBDDA* have substantially lower performance than GHSETA*. In this domain, however, A* outperforms GHSETA*. The reason for this is that the Boolean encoding of the domain is very weak. The domain does not contain any single-valued predicates which forces each grounded predicate to be represented by a Boolean variable. Thus, a more sophisticated planning domain analysis than the one used in this experimental evaluation is necessary. Whether an efficient Boolean encoding exists for this domain is out of the scope of this article. It may be observed, however, that by not deleting the *home* predicate of the top card of a home cell when moving a new card to the cell, the predicates *on*, *incell*, *bottomcol*, and *home* become single-valued in the first argument. The encoding of the *on* predicate is similar to the one used in the Blocks World. The reduction, however, is not as efficient as in the Blocks World. In the Blocks World, a block can be on any other block, but in Free Cell, a card can be on at most three other cards: the one it is on initially and the two successors with opposite color. For problem 6, the number of bits in the BDD encoding is only reduced from 199 to 125, and the performance of the BDD-based algorithms is not improved significantly.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Table 8
Results of the Free Cell problems

Algorithm	p	t_{total}	t_{rel}	t_{search}	$\|sol\|$	$\|expand\|$	$\|Q\|_{\max}$	it	$\|T\|$
GHSETA*	1	1.1	0.7	0.1	8	221.3	24	19	13192
	2	9.5	3.8	4.5	14	418.5	52	210	30048
	3	16.7	8.5	5.5	18	416.7	61	193	62651
	4	302.8	12.9	282.1	29	1116.5	126	1816	127854
	5	176.9	34.8	133.2	30	1046.1	147	885	215259
	6	*Mem*							
BDDA*	1	2.8	1.2	0.8	8	518.8		16	3735
	2	70.1	5.9	62.2	14	3022.7		60	8933
	3	110.4	15.6	91.0	18	1871.9		125	17153
	4	*Mem*							
	5	*Mem*							
	6	*Mem*							
iBDDA*	1	1.6	0.7	0.3	8	510.8		16	3735
	2	30.2	3.6	23.9	14	3014.7		60	8933
	3	51.3	5.7	32.1	18	1863.9		125	26009
	4	*Mem*							
	5	*Mem*							
	6	*Mem*							
A*	1	0.5		0.4	8			138	
	2	14.7		14.5	14			2193	
	3	11.2		11.0	18			1254	
	4	431.3		422.9	29			37655	
	5	214.1		212.7	30			12190	
	6	939.2		926.9	35			35920	
BIDIR	1	1.1	0.7	0.1	8	1687.1		8	3735
	2	7.7	3.6	2.8	14	22252.1		14	8933
	3	39.7	5.6	31.1	18	92794.4		18	25136
	4	418.7	7.7	402.4	26	389276.0		26	58746
	5	*Mem*							
	6	*Mem*							

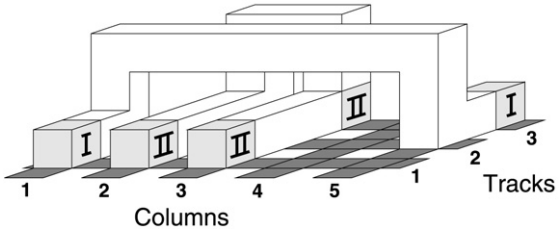

Fig. 23. A solution to a channel routing problem with 5 columns, 3 tracks, and 2 nets (labeled I and II). The pins are numbered according to what net they belong.

7.5. Channel routing

Channel routing is a fundamental subtask in the layout process of VLSI-design. It is an NP-complete problem which makes exact solutions hard to produce. Channel routing considers connecting pins in the small gaps or channels between the cells of a chip. In its classical formulation two layers are used for the wires: one where wires go horizontal (tracks) and one where wires go vertical (columns). In order to change direction, a connection must be made between the two layers. These connections are called vias. Pins are at the top and bottom of the channel. A set of pins that must be connected is called a net. The problem is to connect the pins optimally according to some cost function. The

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

Table 9
Results of the ISCAS-85 channel routing problems. A problem, c–t–n, is identified by its number of columns (c), tracks (t), and nets (n)

Circuit	c–t–n	t_{total}	t_{rel}	t_{search}	$\|Q\|_{\max}$	it
Add	38–3–10	0.2	0.1	0.2	1	40
	47–5–27	0.8	0.7	0.1	24	46
	41–3–12	0.2	0.1	0.1	1	42
	46–7–20	5.0	3.5	1.5	56	89
	25–4–6	0.1	0.0	0.1	1	30
C432	83–4–33	0.4	0.2	0.2	0	93
	89–11–58	*Mem*				
	101–9–57	286.1	61.5	206.6	135	113
	99–8–58	34.0	13.5	20.5	59	448
	97–10–63	295.0	99.7	195.3	129	109
	101–7–53	15.7	11.5	4.2	90	101
	95–9–48	223.8	58.9	164.9	59	399
	95–10–48	*Time*				
	84–5–23	3.2	0.7	2.5	0	92

cost function studied here equals the total number of vias used in the routing. Fig. 23 shows an example of an optimal solution to a small channel routing problem. The cost of the solution is 4. One way to apply search to solve a channel routing problem is to route the nets from left to right. A state in this search is a column paired with a routing of the nets on the left side of that column. A transition of the search is a routing of live nets over a single column. A* can be used in the usual way to find optimal solutions. An admissible heuristic function for our cost function is the sum of the cost of routing all remaining nets optimally ignoring interactions with other nets. We have implemented a specialized search engine to solve channel routing problems with GHSETA* [31]. The important point about this application is that GHSETA* utilizes a conjunctive branching partitioning instead of a disjunctive branching partitioning as in all other experiments reported in this article. This is possible since a transition can be regarded as the joint result of routing each net in turn.

The performance of GHSETA* is evaluated using problems produced from two ISCAS-85 circuits [57]. For each of these problems the parameters of the BDD package are hand tuned for best performance. There is no upper bound on the size of BDDs in frontier nodes ($u = \infty$) and no limit on the size of the partitions. Time out is 600 seconds. Table 9 shows the results. The performance of GHSETA* is similar to previous applications of BDDs to channel routing [54, 57]. However, in contrast to previous approaches, GHSETA* is able to find optimal solutions.

7.6. Additional comparative experiments

The major challenge for BDDA* is that the arithmetic computations at the BDD level scales poorly with the size of the BDD representing the set of states to expand (lines 5 and 6 in Fig. 14). This hypothesis can be empirically verified by measuring the CPU time used by FSETA* and iBDDA* to expand a set of states. Recall that both FSETA* and iBDDA* expand the exact same set of states in each iteration. Any performance difference is therefore solely caused by their expansion techniques. The results are shown in Fig. 24. The reported CPU time is the average of the 15-puzzle with 50, 100, and 200 random steps, Logistics problem 4 to 9, Blocks World problem 4 to 9, Gripper problem 1 to 20, and $D^x V^4 M^{15}$ with x varying from 1 to 6. For very small frontier BDDs, iBDDA* is slightly faster than FSETA*. This is probably because small frontier BDDs mainly are generated by easy problems where a monolithic transition relation used by iBDDA* is more efficient than the partitioned transition relation used by FSETA*. However, for large frontier BDDs, iBDDA* needs much more expansion time than FSETA*.

7.7. State-set branching versus single-state heuristic search

Heuristic search is trivial if the heuristic function is very informative. In this case, state-set branching may have worse performance than single-state heuristic search due to the overhead of computing the transition relation. We have seen an example of this in problem 8 of the Pipes World. Moreover, if we use an inefficient Boolean encoding of a domain, the performance of state-set branching may deteriorate and become worse than single-state heuristic search

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

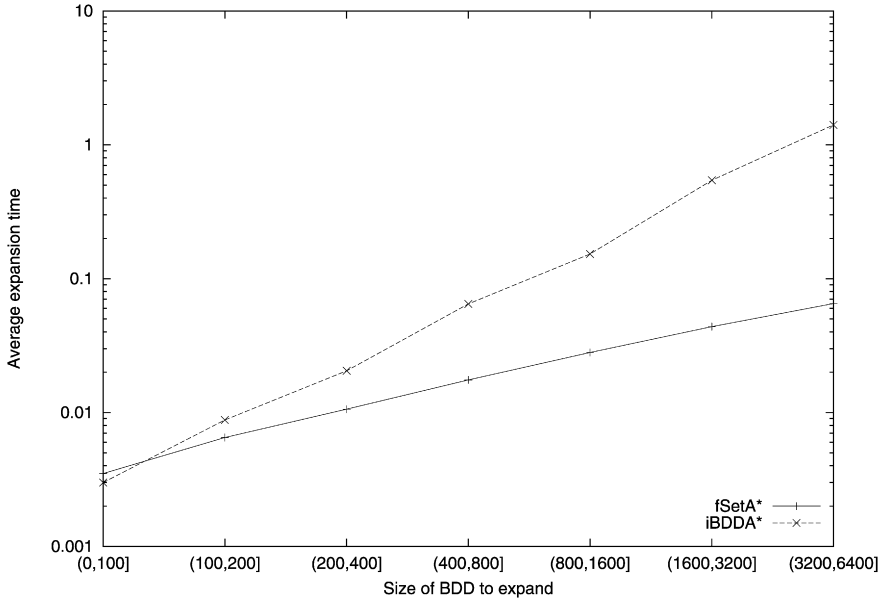

Fig. 24. Node expansion times of FSETA* and BDDA*.

as we have seen for the Free Cell domain. When neither of these issues are present, however, state-set branching has outperformed single-state heuristic search in our experiments.

In order to control the experimental setting and in particular use the same heuristics for all algorithms, we have handcoded our single-state A* implementations. Thus, we are not using state-of-the-art implementations of single-state A* and our experiments do not show that state-set branching can outperform state-of-the-art implementations of single-state A*. Only a direct comparison can verify this.

On the other hand, specialized versions of single-state A* that take advantage of the domain structure have been developed for $\mathbf{FG}^k$, $D^x V^y M^z$, and the $(N^2 - 1)$-puzzles. The state representation used by single-state A* in these domains is at the same level of sophistication as the node representation used by the BuDDy package. For that reason, we consider the comparison in these domains quite fair. For the planning domains, on the other hand, the general representation of states as sets of facts may be improved by using representation techniques from the planning community that we are not aware of. This may reduce the memory consumption of single-state A* by a factor. However, no matter which explicit state representation is used, the space consumption of a set of states will be linear in the size of the set.

Another issue when comparing state-set branching and single-state heuristic search is whether the heuristic function can be chosen as freely for state-set branching as for single-state heuristic search. As described earlier, all of the heuristics applied in our experimental evaluation can be represented compactly by a branching partitioning. This is probably the case for most additive heuristics like the sum of Manhattan distances and HSPr. It is less clear, however, that compact branching partitionings exist for sophisticated heuristics that may have a combinatorial nature or may cover many special cases in an irregular way. Indeed in a recent study, a disjunctive branching partitioning of the max-pair heuristic [28] turned out to be prohibitively large in some domains [32]. We believe that the reason for this is the artificial and combinatorial nature of the max-pair heuristic. The size of the branching partitioning could be dramatically reduced, however, by making it a function of the h-cost of the search node to expand. Developing this kind of representation techniques for complex heuristics is an interesting direction for future work.

7.8. State-set branching versus blind BDD-based search

Blind BDD-based search has been successfully applied in symbolic model checking and circuit verification. It has been shown that many problems encountered in practice are tractable when using BDDs [58]. The classical search problems studied in AI, however, seems to be harder and have longer solutions than the problems considered in formal verification. When applying blind BDD-based search to these problems, the BDDs used to represent the search

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

frontier often grow exponentially. The experimental evaluation of state-set branching shows that this problem can be substantially reduced when efficiently splitting the search frontier according to a heuristic evaluation of the states.

7.9. State-set branching versus BDDA*

State-set branching implementations of A* such as GHSETA* and FSETA* are fundamentally different from BDDA*. BDDA* imitates the usual explicit application of the heuristic function via a symbolic computation. It would be reasonable to expect that the symbolic representation of practical heuristic functions often is very large. However, this is seldom the case for the heuristic functions studied in this article. The major challenge for BDDA* is that the arithmetic computations at the BDD level scales poorly with the size of the BDD representing the set of states to expand (lines 5 and 6 in Fig. 14). Another limitation of BDDA* is the inflexibility of BDD-based arithmetic. It makes it hard to extend BDDA* efficiently to general evaluation functions and arbitrary transitions costs.

8. Conclusion

In this article, we have presented a framework called state-set branching for integrating symbolic and heuristic search. The key component of the framework is a new BDD technique called branching partitioning that allows sets of states to be expanded implicitly and sorted according to cost estimates in one step. State-set branching is a general framework. It applies to any heuristic function, any search node evaluation function, and any transition cost function defined over a finite domain. An extensive experimental evaluation of state-set branching proves it to be a powerful approach. Except for one case with a weak Boolean encoding, state-set branching outperforms single-state heuristic search. In addition, it can improve the complexity of single-state search exponentially and for several of the best-known AI search problems, it is often orders of magnitude faster than single-state heuristic search, blind BDD-based search, and the most efficient current BDD-based A* implementation, BDDA*.

It is an important direction for future work to develop techniques for representing branching partitionings compactly for sophisticated heuristics that may have a combinatorial nature or may cover many special cases in an irregular way. Other directions for future work include applying state-set branching to regression search and linear space heuristic search algorithms such as IDA*.

Acknowledgements

We thank Robert Punkunus for initial work on efficient Boolean encoding of PDDL domains. We also wish to thank Kolja Sulimma for providing channel routing benchmark problems. Finally, we thank our anonymous reviewers for their valuable comments and suggestions.

Appendix A

Lemma 6. *The search structure build by the BSFS algorithm is a DAG where every node $\langle S', \vec{e}' \rangle$ different from a root node $\langle \{s_0\}, \vec{e}_0 \rangle$ has a set of predecessor nodes. For each state $s' \in S'$ in such a node, there exists a predecessor $\langle S, \vec{e} \rangle$ with a state $s \in S$ such that $\mathcal{T}(s, s')$ and $\vec{e}' = \vec{e} + \delta\vec{e}(s, s')$.*

Proof. By induction on the number of loop iterations. We get that the search structure after the first iteration is a DAG consisting of a root node $\langle \{s_0\}, \vec{e}_0 \rangle$. For the inductive step, assume that the search structure is a DAG with the desired properties after n iterations of the loop (see Fig. 5). If the algorithm in the next iteration terminates in line 3 or 5, the search structure is unchanged and therefore a DAG with the required format. Assume that the algorithm does not terminate and that $\langle S, \vec{e} \rangle$ is the node removed from the top of *frontier*. The node is expanded by forming child nodes with the STATESETEXPAND function in line 6. According to the definition of this function, any state $s' \in S'$ in a child node $\langle S', \vec{e}' \rangle$ has some state $s \in S$ in $\langle S, \vec{e} \rangle$ such that $\mathcal{T}(s, s')$ and $\vec{e}' = \vec{e} + \delta\vec{e}(s, s')$. Thus $\langle S, \vec{e} \rangle$ is a valid predecessor for all states in the child nodes. Furthermore, since all child nodes are new nodes, no cycles are created in the search structure which therefore remains a DAG. If a child node is merged with an old node when enqueued on *frontier* the resulting search structure is still a DAG because all nodes on *frontier* are unexpanded and therefore

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

have no successor nodes that can cause cycles. In addition, each state in the resulting node obviously has the required predecessor nodes. □

Lemma 7. *For each state $s' \in S'$ of a node $\langle S', \vec{e}' \rangle$ in a finite search structure of the BSFS algorithm there exists a path $\pi = s_0, \ldots, s_n$ in $\mathcal{D}$ such that $s_n = s'$ and $\vec{e} = \vec{e}_0 + \sum_{i=0}^{n-1} \delta\vec{e}(s_i, s_{i+1})$.*

Proof. We will construct π by tracing the edges backwards in the search structure. Let $b_0 = s'$. According to Lemma 6 there exists a predecessor $\langle S, \vec{e} \rangle$ to $\langle S', \vec{e}' \rangle$ such that for some state $b_1 \in S$ we have $\mathcal{T}(b_1, b_0)$ and $\vec{e}' = \vec{e} + \delta\vec{e}(b_1, b_0)$. Continuing the backward traversal from b_1 must eventually terminate since the search structure is finite and acyclic. Moreover, the traversal will terminate at the root node because this is the only node without predecessors. Assume that the backward traversal terminates after n iterations. Then $\pi = b_n, \ldots, b_1$. □

Theorem 8. *The BSFS algorithm is sound.*

Proof. Assume that the algorithm returns a path $\pi = s_0, \ldots, s_n$ with cost estimates $\vec{e}$. Since $s_n \in \mathcal{G}$ it follows from Lemma 7 and the definition of EXTRACTSOLUTION that π is a solution to the search problem associated with cost estimates $\vec{e}$. □

Lemma 9. *Assume* FSETA* *and* GHSETA* *apply an admissible heuristic and $\pi = s_0, \ldots, s_n$ is an optimal solution, then at any time before* FSETA* *and* GHSETA* *terminates there exists a frontier node $\langle S, \vec{e} \rangle$ with a state $s_i \in S$ such that $\vec{e} \leqslant C^*$ and $s_0, \ldots, s_i$ is the search path associated with s_i.*

Proof. A node $\langle S, \vec{e} \rangle$ containing s_i with associated search path $s_0, \ldots, s_i$ must be on the frontier since a node containing s_0 was initially inserted on the frontier and FSETA* and GHSETA* terminates if a node containing the goal state s_n is removed from the frontier. We have $\vec{e} = cost(s_0, \ldots, s_i) + h(s_i)$. The path $s_0, \ldots, s_i$ is a prefix of an optimal solution, thus $cost(s_0, \ldots, s_i)$ must be the minimum cost of reaching s_i. Since the heuristic function is admissible, we have $h(s_i) \leqslant h^*(s_i)$ which gives $\vec{e} \leqslant C^*$. □

Theorem 10. *Given an admissible heuristic function* FSETA* *and* GHSETA* *are optimal.*

Proof. Suppose FSETA* or GHSETA* terminates with a solution derived from a frontier node with $\vec{e} > C^*$. Since the node was at the top of the frontier queue, we have

$$C^* < f(n) \quad \forall n \in \mathit{frontier}.$$

However, this contradicts Lemma 9 that states that any optimal path has a node on the frontier any time prior to termination with $\vec{e} \leqslant C^*$. □

References

[1] S.B. Akers, Binary decision diagrams, IEEE Transactions on Computers C-27 (6) (1978) 509–516.
[2] F. Bacchus, AIPS-00 Planning Competition: The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems, AI Magazine 22 (3) (2001) 47–56.
[3] R. Bahar, E. Frohm, C. Gaona, E Hachtel, A Macii, A. Pardo, F. Somenzi, Algebraic decision diagrams and their applications, in: IEEE/ACM International Conference on CAD, 1993, pp. 188–191.
[4] A. Barrett, D.S. Weld, Partial-order planning: Evaluating possible efficiency gains, Artificial Intelligence 67 (1) (1994) 71–112.
[5] P. Bertoli, A. Cimatti, M. Roveri, Conditional planning under partial observability as heuristic-symbolic search in belief space, in: Proceedings of the 6th European Conference on Planning (ECP-01), 2001, pp. 379–384.
[6] P. Bertoli, M. Pistore, Planning with extended goals and partial observability, in: Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04), 2004, pp. 270–278.
[7] R. Bloem, K. Ravi, F. Somenzi, Symbolic guided search for CTL model checking, in: Proceedings of the 37th Design Automation Conference (DAC-00), ACM, 2000, pp. 29–34.
[8] B. Bonet, H. Geffner, Planning as heuristic search: New results, in: Proceedings of the 5th European Conference on Planning (ECP-99), Springer, 1999, pp. 360–372.
[9] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers 8 (1986) 677–691.
[10] D. Bryce, D.E. Smith, Planning graph heuristics for belief space search, Journal of Artificial Intelligence Research 26 (2006) 35–99.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

[11] J.R. Burch, E.M. Clarke, D.E. Long, Symbolic model checking with partitioned transition relations, in: International Conference on Very Large Scale Integration, North-Holland, 1991, pp. 49–58.
[12] A. Cimatti, E. Giunchiglia, F. Giunchiglia, P. Traverso, Planning via model checking: A decision procedure for $\mathcal{AR}$, in: Proceedings of the 4th European Conference on Planning (ECP-97), Springer, 1997, pp. 130–142.
[13] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking, Artificial Intelligence 147 (1–2) (2003).
[14] A. Cimatti, M. Roveri, Conformant planning via symbolic model checking, Journal of Artificial Intelligence Research 13 (2000) 305–338.
[15] E. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.
[16] S. Edelkamp, Directed symbolic exploration in AI-planning, in: AAAI Spring Symposium on Model-Based Validation of Intelligence, 2001, pp. 84–92.
[17] S. Edelkamp, Symbolic exploration in two-player games: Preliminary results, in: Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-02) Workshop on Model Checking, 2002.
[18] S. Edelkamp, Symbolic pattern databases in heuristic search planning, in: Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS-02), 2002, pp. 274–283.
[19] S. Edelkamp, External symbolic heuristic search with pattern databases, in: Proceedings of the 15th International Conference on AI Planning and Scheduling (ICAPS-05), 2005, pp. 51–60.
[20] S. Edelkamp, M. Helmert, Exhibiting knowledge in planning problems to minimize state encoding length, in: Proceedings of the 6th European Conference on Planning (ECP-99), 1999, pp. 135–147.
[21] S. Edelkamp, F. Reffel, OBDDs in heuristic search, in: Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI-98), Springer, 1998, pp. 81–92.
[22] R.E. Fikes, N.J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (1971) 189–208.
[23] M.P. Fourman, Propositional planning, in: Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning, 2000, pp. 10–17.
[24] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains, Journal of Artificial Intelligence Research (JAIR) 20 (2003) 61–124.
[25] A. Gerevini, L. Schubert, Inferring state constraints for domain-independent planning, in: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98), 1998, pp. 905–912.
[26] E. Hansen, R. Zhou, Z. Feng, Symbolic heuristic search using decision diagrams, in: Symposium on Abstraction, Reformulation and Approximation SARA-02, 2002.
[27] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for heuristic determination of minimum path cost, IEEE Transactions on SSC 100 (4) (1968).
[28] P. Haslum, H. Geffner, Admissible heuristics for optimal planning, in: Proceedings of the 5th International Conference on Artificial Intelligence Planning System (AIPS-00), AAAI Press, 2000, pp. 140–149.
[29] J. Hoffmann, S. Edelkamp, The deterministic part of IPC-4: An overview, Journal of Artificial Intelligence Research (JAIR) 24 (2005) 519–579.
[30] R.M. Jensen, R.E. Bryant, M.M. Veloso, SetA*: An efficient BDD-based heuristic search algorithm, in: Proceedings of 18th National Conference on Artificial Intelligence (AAAI-02), 2002, pp. 668–673.
[31] R.M. Jensen, R.E. Bryant, M.M. Veloso, SetA* applied to channel routing. Technical report, Computer Science Department, Carnegie Mellon University, 2002. CMU-CS-02-172.
[32] R.M. Jensen, E.A. Hansen, S. Richards, R. Zhou, Memory-efficient symbolic heuristic search, in: Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06), 2006, pp. 304–313.
[33] R.M. Jensen, M.M. Veloso, OBDD-based deterministic planning using the UMOP planning framework, in: Proceedings of the AIPS-00 Workshop on Model-Theoretic Approaches to Planning, 2000, pp. 26–31.
[34] R.M. Jensen, M.M. Veloso, OBDD-based universal planning for synchronized agents in non-deterministic domains, Journal of Artificial Intelligence Research 13 (2000) 189–226.
[35] R.M. Jensen, M.M. Veloso, M. Bowling, Optimistic and strong cyclic adversarial planning, in: Proceedings of the 6th European Conference on Planning (ECP-01), 2001, pp. 265–276.
[36] R.M. Jensen, M.M. Veloso, R.E. Bryant, Fault tolerant planning: Toward probabilistic uncertainty models in symbolic non-deterministic planning, in: Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04), 2004, pp. 335–344.
[37] U. Kuter, D. Nau, M. Pistore, P. Traverso, A hierarchical task-network planner based on symbolic model checking, in: Proceedings of the 15th International Conference on Automated Planning and Scheduling ICAPS-05, 2005, pp. 300–309.
[38] J. Lind-Nielsen, BuDDy—A Binary Decision Diagram Package, Technical Report IT-TR: 1999-028, Institute of Information Technology, Technical University of Denmark, 1999, http://sourceforge.net/projects/buddy.
[39] D. Long, M. Fox, The AIPS-02 planning competition, http://planning.cis.strath.ac.uk/competition/, 2002.
[40] D. Long, H.A. Kautz, B. Selman, B. Bonet, H. Geffner, J. Koehler, M. Brenner, J. Hoffmann, F. Rittinger, C.R. Anderson, D.S. Weld, D.E. Smith, M. Fox, The AIPS-98 planning competition, AI Magazine 21 (2) (2000) 13–33.
[41] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL—the planning domain definition language, Technical report, Yale Center for Computational Vision and Control, 1998.
[42] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publ., 1993.
[43] C. Meinel, C. Stangier, A new partitioning scheme for improvement of image computation, in: Proceedings ASP-DAC'2001, 2001, pp. 97–102.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009

[44] C. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design, Springer, 1998.
[45] A. Nymeyer, K. Qian, Heuristic search algorithms based on symbolic data structures, in: Proceedings of the 16th Australian Conference on Artificial Intelligence, in: Lecture Notes in Computer Science, vol. 2903, Springer, 2003, pp. 966–979.
[46] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984.
[47] M. Pistore, R. Bettin, P. Traverso, Symbolic techniques for planning with extended goals in non-deterministic domains, in: Proceedings of the 6th European Conference on Planning (ECP-01), 2001, pp. 253–264.
[48] I. Pohl, First results on the effect of error in heuristic search, Machine Intelligence 5 (1970) 127–140.
[49] K. Qian, A. Nymeyer, Guided invariant model checking based on abstraction and symbolic pattern databases, in: Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS-04), 2004, pp. 497–511.
[50] R.K. Ranjan, A. Aziz, R.K. Brayton, B. Plessier, C. Pixley, Efficient BDD algorithms for FSM synthesis and verification, in: IEEE/ACM Proceedings of the International Workshop on Logic Synthesis, 1995.
[51] F. Reffel, S. Edelkamp, Error detection with directed symbolic model checking, in: Proceedings of World Congress on Formal Methods (FM), Springer, 1999, pp. 195–211.
[52] D. Sawitzki, Experimental studies of symbolic shortest-path algorithms, in: Proceedings of the 3rd International Workshop on Experimental and Efficient Algorithms (WEA-04), 2004, pp. 482–498.
[53] D. Sawitzki, A symbolic approach to the all-pairs shortest-paths problem, in: Proceedings of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG-04), 2004, pp. 154–168.
[54] F. Schmiedle, R. Drechsler, B. Becker, Exact channel routing using symbolic representation, in: Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS-99), 1999.
[55] F. Somenzi, CUDD: Colorado University Decision Diagram Package, ftp://vlsi.colorado.edu/pub/, 1996.
[56] H.-P. Störr, Planning in the fluent calculus using binary decision diagrams, AI Magazine (2001) 103–105.
[57] K. Sulimma, W. Kunz, An exact algorithm for solving difficult detailed routing problems, in: Proceedings of the 2001 International Symposium on Physical Design, 2001, pp. 198–203.
[58] I. Wegener, Branching Programs and Binary Decision Diagrams, Society for Industrial and Applied Mathematics (SIAM), 2000.
[59] C.H. Yang, D.L. Dill, Validation with guided search of the state space, in: Proceedings of the 35th Design Automation Conference (DAC-98), ACM, 1998, pp. 599–604.
[60] J. Yuan, J. Shen, J. Abraham, A. Aziz, Formal and informal verification, in: Conference on Computer Aided Verification (CAV-97), 1997, pp. 376–387.

Please cite this article in press as: R.M. Jensen et al., State-set branching: Leveraging BDDs for heuristic search, Artificial Intelligence (2007), doi:10.1016/j.artint.2007.05.009